MASTER EN SCIENCES INFORMATIQUES

Etude de la qualité des applications de bases de données

Lemaitre, Jonathan

Award date:
2007

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Étude de la qualité des applications de bases de données

Jonathan Lemaitre

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique
Année Académique 2006 - 2007

Préface et remerciements

Ce mémoire s'inscrit dans deux domaines distincts qui n'ont jusqu'à présent eu que peu d'interactions. L'idée de départ a été donnée par les Professeurs Alain April et Jean-Luc Hainaut qui ont abordé ce problème il y a presque dix ans. Peu d'éléments directement liés à ce sujet sont venus se rajouter depuis dans la littérature. On peut donc encore qualifier ce problème d'inexploré. Ce mémoire est en fait le développement d'une idée, développement qui s'est déroulé sur deux continents différents. Celui-ci a commencé à Montréal, à l'École de Technologie Supérieure (ETS). Sa rédaction s'est ensuite poursuivie à l'Institut d'Informatique, aux Facultés Universitaires Notre-Dame de la Paix, à Namur.

Je tiens à remercier les Professeurs Alain April et Jean-Marc Desharnais, de l'ETS, pour l'aide qu'ils ont apportée sur ce mémoire et pour leur expertise dans les mesures de taille fonctionnelle.

De l'Institut d'Informatique, je remercie le Professeur Jean-Luc Hainaut, mon Promoteur et Professeur, pour ses idées et son soutient par rapport à ce mémoire. J'adresse également un remerciement particulier au Professeur Naji Habra pour ses connaissances en qualité du logiciel et à Anthony Clève pour toute l'aide qu'il m'a apportée et toutes ses critiques qui m'ont permis d'avancer.

Finalement, je remercie Audrey et Frédéric pour leurs précieux conseils, leur soutient et bien d'autres choses encore.

Résumé

Ce mémoire aborde le problème de la qualité du logiciel et plus particulièrement de sa maintenabilité. Ce thème, déjà traité par de nombreux auteurs, est analysé ici dans une approche orientée données. Nous verrons comment le schéma des données, la base de données et les accès à la base de données peuvent, au travers de différentes techniques, apporter de nouvelles informations sur la qualité du logiciel. Ces informations pourront par la suite être utilisées durant un processus de maintenance. Ce mémoire se situe donc au croisement de l'ingénierie du logiciel et des bases de données. La croisée de ces deux domaines n'a que peu de fois été explorée depuis la mise en évidence du problème de la qualité du logiciel. Nous présenterons dans un premier temps quelques techniques et modèles existants. Ces techniques et modèles seront soit directement liés au sujet traité, soit basés sur un concept similaire à l'analyse des données. Dans un second temps, nous développerons le problème au travers de nouvelles techniques et de nouveaux outils destinés à l'analyse de la qualité du logiciel. Au fil des développements, nous tenterons d'établir des liens concrets entre les outils développés et les tâches de maintenance. Nous illustrerons également certaines théories par une mise en pratique et des études de cas.

Abstract

This thesis addresses the problem of the software quality and particularly of the maintainability. These two subjects have already been analyzed by many authors. However they will be analyzed here under a data-centred approach. We will see how the data schemas, the databases and the access to the databases can bring new information on the software quality. For that, we will have to develop many different tools, which will give us new information. Thereafter this new information could be used during maintenance's process. Then this thesis lies in the cross-road between the software engineering and the databases. This two domains' intersection was rarely studied since the underlining of the software quality problem. In the first time we will present some existing techniques and models. Secondly we will develop the problem trough new techniques and tools, bounded to analyse the software quality. Throughout these developments we will attempt to establish concrete links between the developed tools and the maintenance's tasks. We will also illustrate some theories trough practice and case studies.

Table des matières

Table des figures

Chapitre 1

Introduction

Depuis les années ’80, le problème de la qualité du logiciel a tenu une part de plus en plus importante dans les différents processus liés au logiciel, comme le développement, la maintenance, etc. Les principes de la qualité du logiciel, regroupés dans le domaine de l’ingénierie du logiciel, ont pour buts d’améliorer, d’estimer et de contrôler la qualité de l’application. A l’instar de la qualité des processus de développement, celle du logiciel sous-tend un but économique. Par exemple, un client, pour des raisons internes à son entreprise, souhaite obtenir une application répondant à certains critères de performance. Les développeurs, quant à eux, souhaitent pouvoir prévoir le coût de la maintenance. Les caractéristiques liées à la qualité d’un logiciel sont donc devenues presque aussi importantes que les fonctionnalités qu’il implémente.

Bien sûr, ces caractéristiques sont nombreuses et ont été analysées et mesurées de multiples façons depuis la création de l’ingénierie des logiciels. Toutefois, il semble que les différents auteurs se soient principalement concentrés sur ce que pouvait apporter les documents de conception de l’application et sur le code de celle-ci. Les ouvrages se rapportant à l’influence que peuvent avoir les données, leur structure et les accès qui y sont faits, sont encore peu nombreux à l’heure actuelle. Nous souhaitons ici explorer cette direction afin d’améliorer les connaissances se rapportant à la qualité du logiciel. Bien évidemment, les caractéristiques abordées seront limitées en nombre et nous nous concentrerons sur le concept de maintenabilité. Notre étude fera appel à des éléments tels que la taille, la complexité et la cohésion.

La maintenabilité exprime la facilité avec laquelle un processus de maintenance peut être effectué. Elle est encore de nos jours une des caractéristiques des logiciels les plus difficiles à évaluer. Elle est également une des plus importantes. Cette importance a été établie dans différentes études faisant état des problèmes et des coûts de la maintenance. Jusqu’il y a une dizaine d’années, l’effort consacré à la maintenance n’a cessé de croître en termes de temps et de coûts. Dans les années ’80, Boehm [8] présentait déjà l’importance de les contrôler tout en rendant une meilleure qualité générale au logiciel et cela afin d’éviter la mobilisation inutile de ressources. Par après, différentes études ont confirmé cette idée et ont mis en évidence l’importance de la qualité. Ces études ont fait état de prédictions et de réalités sur les coûts de la maintenance. Coleman [13] et Cheaito [9], en regroupant d’autres études, ont estimé ces coûts. Coleman estimait le coût de la maintenance entre 40% et 60% des coûts de développement. Cheaito annonçait que, en moyenne, une entreprise développant des logiciels utilisait 60% de ses ressources pour la maintenance. Cette tendance s’est confirmée et ces coûts ont atteint jusqu’à 75% du budget alloué au projet [42].

Le contrôle des coûts de maintenance est donc important. Ce contrôle demande de

vérifier la qualité de l'application, entre autres avec l'aide d'outils et de théories de l'ingénierie du logiciel se rapportant à la maintenabilité (ex : [36]). Nous souhaitons, dans ce mémoire, développer une base théorique accompagnée d'outils et de techniques, permettant de comprendre d'où proviennent les difficultés qui vont être rencontrées durant la maintenance, de quantifier si possible ces difficultés et de pouvoir comparer différentes parties du logiciel selon leur niveau de qualité. Nous souhaitons donc pouvoir identifier, dans un logiciel, les éléments pouvant entraîner un problème de maintenance. Nous espérons apporter de nouvelles réponses en abordant le problème différemment. Notre but n'est pas d'améliorer explicitement les méthodes de design et d'implémentation des systèmes. Nous n'aborderons en effet que les problèmes liés aux logiciels déjà développés. Les techniques que nous développerons appartiendront principalement au domaine des métriques. Les concepts de métrique, modèle et mesure seront définis dans la section suivante. Les outils développés feront appel à la structure des données, c'est-à-dire un schéma conceptuel, logique ou physique, ainsi qu'aux accès aux données par des requêtes ou par du code procédural.

1.1 Introduction générale au concept de métrique

Le terme "métrique", bien qu'évoquant intuitivement la mesure de quelque chose, reste généralement assez abstrait. C'est pourquoi nous commencerons par définir ce que sont les métriques en génie logiciel. En plus de cela, nous verrons les grands principes et les normes de qualité du logiciel utilisés dans ce mémoire. Les concepts suivants proviennent de la littérature et en particulier de l'ouvrage de Fenton et Pfleeger [15] dans lequel sont rassemblés et détaillés des modèles et mesures de qualité, des techniques, etc.

1.1.1 Définition générale et liste de normes

Les métriques logicielles appartiennent à la branche de l'ingénierie du logiciel. Cette dernière regroupe l'ensemble des techniques qui appliquent une approche d'ingénieur à la construction et au support des produits logiciels. Elle se concentre sur l'implémentation de logiciels d'une manière contrôlée et scientifique. Le contrôle implique évidemment d'avoir à sa disposition un ensemble de techniques, mesures, modèles, etc., afin de pouvoir observer et influencer les caractéristiques du processus étudié. Le terme scientifique sous-entend une base théorique et réfléchie qui permet de justifier les outils utilisés. Les processus de mesures sont donc nécessaires dans ce contexte. Ils permettent d'observer les caractéristiques de l'objet étudié, d'évaluer les relations entre ces caractéristiques et la qualité du logiciel et de décrire une solution au problème observé ou au contraire de constater l'absence de problème.

Le terme métrique, en génie logiciel, est difficile à définir. En effet, la portée de ce terme s'étend dans de nombreuses directions. Fenton et Pfleeger utilisent la définition suivante :

"... Software metrics is a term that embraces many activities, all of which involve some degree of software measurement :

- *Cost and effort estimation*
- *Productivity measures and models*
- *Data collection*
- *... "*

Les métriques de l'ingénierie du logiciel regroupent donc de nombreuses activités qui

impliquent toutes à certains niveaux, un processus de mesure du logiciel [1]. Ces activités sont par exemple :

- des estimations du coût et de l'effort ;
- des mesures et des modèles de productivité ;
- le rassemblement de données ;
- des mesures et modèles de qualité ;
- des modèles de fiabilité ;
- des évaluations et modèles de performance ;
- des outils d'évaluation de la structure et de la complexité d'un logiciel ;
- des estimations de capacité et de niveau de compétence ;
- des évaluations de méthodes et d'outils ;
- ...

Le terme métrique est toutefois utilisé pour désigner d'autres concepts particuliers englobés par la définition ci-dessus. Il est souvent utilisé pour désigner le terme de mesure, ce qui peut se justifier par le fait que le concept de mesure est prédominant. L'importance du fait de mesurer est souligné dans la définition générale des métriques et dans celle du génie logiciel. Il peut également désigner le modèle qui, sur base du résultat de la mesure de certains attributs du logiciel, permet de contrôler certains aspects de l'objet mesuré, de situer les changements nécessaires, de conseiller certaines modifications de l'objet, de prédire les états futurs, etc. Par exemple, la complexité définie par McCabe, que nous aborderons dans les parties suivantes, constitue un modèle d'évaluation de la complexité, souvent désigné comme une métrique. De même, un modèle de prédiction du nombre de lignes de code en fonction de la taille du schéma des données est parfois appelé métrique. Nous garderons donc la terminologie suivante, le terme mesure se rapportera à l'évaluation de la valeur d'un attribut direct de l'objet analysé. Le résultat sera donc la valeur d'une caractéristique de l'objet. Le terme modèle se rapportera à un ensemble d'opérations et de formules qui vont faire intervenir le résultat d'une ou plusieurs mesures afin de prédire un aspect du logiciel.

1.1.2 Enjeux des métriques

Le coût d'un logiciel est bien entendu l'un des enjeux principaux pour tous les processus liés au logiciel, autant pour le client que pour le développeur. Ce coût a été un des premiers aspects étudiés en génie logiciel. Néanmoins, la qualité du logiciel est également fondamentale et fait maintenant pleinement partie des exigences du client. Certaines exigences de qualité sont demandées par le client, comme par exemple la performance ou la facilité d'utilisation des interfaces. D'autres qualités sont plus spécifiques aux développeurs, comme la réutilisabilité ou la maintenabilité. Elles ont un impact à plus long terme mais sont également nécessaires.

Comme le présente Boehm [8], le contrôle de la qualité n'implique pas nécessairement une perte en termes d'efforts et de coûts. L'absence d'un contrôle engendre une économie en termes d'efforts mais représente également un risque qui peut être significatif, comme l'ont montré les différentes études réalisées sur les résultats de projets de développement [2], [32], [33] ainsi que celles sur le coût de la maintenance [35], [6], [32]. La prise en compte de la qualité peut, par contre, apporter des informations vitales lors du développement et permettre d'éviter un échec. L'enjeu des métriques se situe donc à ce niveau.

Le coût nécessaire à la mise en place de modèles et mesures varie mais peut être mi-

[1] Le terme logiciel est utilisé au sens large, c'est-à-dire qu'il reprend le système, les divers documents et toutes les données liées au projet, que ce soit pendant le développement, pendant un processus de maintenance ou encore durant l'utilisation du système.

nime si ceux-ci sont intégrés dans des outils, ou encore *CASE tool*[2]. L'automatisation des mesures et modèles constitue un problème que beaucoup tentent évidemment de résoudre. C'est par exemple le cas lorsque la mesure fait intervenir une évaluation dont les résultats varient d'un expert à l'autre, ou encore une caractéristique externe au logiciel, qui change en fonction de l'environnement. Un outil ne peut alors plus évaluer correctement la situation sans une aide extérieure.

La mise en place d'une analyse automatisée, au travers d'un outil, permet de diminuer le coût par rapport à une analyse équivalente réalisée par un expert. Un tel outil fournit également un gain de temps important. Etant donné ces avantages, nous tenterons de ne développer que de nouveaux modèles et mesures, construits de façon à pouvoir être automatisés. Nous verrons également que les résultats de certaines mesures subjectives [3] peuvent être approchés par des modèles et mesures automatisables.

1.1.3 Quelques mesures et modèles

Afin d'illustrer les notions de mesures et modèles, voici quelques brefs exemples parmi les plus connus.

Un logiciel présente plusieurs caractéristiques. Celles-ci peuvent ne dépendre que du logiciel, ou alors être influencées par des éléments externes, comme par exemple la compétence d'un programmeur. Fenton [15] fait souvent l'analogie avec l'être humain, mais cette comparaison peut-être faite avec n'importe quel objet et même avec des choses immatérielles (ex : le temps, la pression atmosphérique, etc.). De nos jours, il semble naturel pour l'homme de pouvoir tout quantifier et mesurer.

Beaucoup de modèles ont été proposés et reconnus dans un premier temps pour être finalement rejetés, soit parce qu'ils n'étaient pas démontrés empiriquement, soit parce qu'ils ont été remplacés par un autre modèle ou encore réfutés. C'est par exemple le cas de la complexité de Halstead [30] qui fait appel à une mesure du nombre de symboles et d'opérateurs se trouvant dans le code d'une application afin d'en déterminer la complexité au travers d'un calcul logarithmique. Cette méthode s'appuie sur les sciences humaines. Une des applications les plus intéressantes de cette mesure a été l'estimation du temps nécessaire à un être humain pour comprendre le code d'une application. Malgré le fait que la mesure était automatisable, la complexité de Halstead fut rejetée par la communauté à cause de son manque de fondement dans l'évaluation de la complexité du code.

D'autres modèles et mesures sont toujours utilisés alors qu'ils sont mis en doute. C'est le cas par exemple de la complexité de McCabe [39] qui caractérise la complexité du logiciel en fonction du nombre de noeuds de décision se trouvant dans les différents modules ou différentes fonctions de l'application. McCabe donne également dans son modèle une valeur de complexité à ne pas dépasser, celle-ci étant de 10. La complexité de McCabe est encore utilisée de nos jours et a été intégrée dans de nombreux modèles. Elle est toutefois remise en question étant donné l'évolution des langages, le manque de validation théorique et les variations des résultats empiriques.

Une mesure classique et toujours d'actualité est le nombre de lignes de code. Cette mesure calcule une taille physique du code. Il existe plusieurs façons de mesurer la taille du code, certains éléments du code ne devant pas être pris en compte, selon les choix des analystes. On peut par exemple considérer tout le fichier ou ne pas compter les lignes vides ainsi que celles de commentaires. On peut également considérer le nombre de lignes comme étant le nombre d'instructions dans le code. Ces choix doivent être faits avant de

[2] CASE (Computer-aided software engineering) tool : Outils de soutien aux processus de développement et/ou de maintenance des logiciels.

[3] Nous considèrerons une mesure comme étant subjective lorsqu'elle peut founir des résultats différents, dans un même environnement et sur le même objet, en fonction des jugements d'experts.

commencer le processus de mesure. Bien que cette mesure soit critiquée, en particulier à cause de la réutilisation de code, modules, classes, etc., elle est toujours couramment utilisée et intervient notamment dans les modèles estimant la productivité des programmeurs et dans l'estimation de certains processus logiciels.

La performance est une qualité du logiciel qui concerne, par exemple, le temps nécessaire à une application pour la réalisation d'une ou plusieurs opérations. Elle peut être une contrainte imposée par le client. Son évaluation peut se faire par estimation ou directement pendant l'exécution du logiciel qui peut être réalisée dans différents milieux (situations réelles d'utilisation du logiciel, milieux fermés, etc.). On peut par exemple mesurer les temps d'accès d'une application à une fonction d'une autre application en passant par un réseau, le temps de lecture dans une base de données ou encore simplement le temps d'exécution d'une fonction du programme. Elle peut donc être mesurée directement ou estimée en utilisant des modèles prédictifs comme c'est le cas par exemple pour des accès à une base de données [27].

1.1.4 Classifications des métriques

Les métriques regroupent des modèles et mesures dont les approches et moments d'utilisation varient fortement. Par exemple, ils peuvent être utilisés au début du développement à partir des premiers documents de design et de conception, afin d'estimer la taille du logiciel, que celle-ci soit en lignes de code ou en terme de fonctions, etc. On peut aussi faire appel à des mesures basées sur le code de l'application. Celles-ci seront alors utilisées après le développement. Le moment auquel elles s'utilisent n'est pas le seul moyen de les différencier.

La sélection d'un ensemble de modèles se fait généralement en fonction des attributs et qualités du logiciel qui doivent être évalués[15], [17], [23]. Un modèle est donc choisi en fonction de ce sur quoi porte son résultat, de l'information qu'il va apporter. Il s'agit de l'approche la plus naturelle, utilisée lorsque l'on sait ce qui doit être mesuré, par exemple, la taille, le nombre de fautes par module, la complexité d'un module, l'efficacité, etc. Ce mémoire sera principalement consacré à la maintenabilité. Les mesures que nous aborderons par après devraient donc permettre de contrôler, d'accroître ou encore de situer les problèmes liés à la maintenabilité.

Les modèles de mesures peuvent également être regroupés en fonction des inputs qui leurs sont nécessaires. Les principaux inputs sont le code, les documents de conception et les données sur le processus de développement. Comme nous l'avons dit précédemment, nous utiliserons le code et les schémas décrivant la structure des données. Il est également possible de classer les mesures en fonction des aspects analysés. Les aspects les plus courants sont ceux concernant les données, les flux et les états. Ces aspects particuliers sont utilisés lors de l'analyse du système, que ce soit au travers du code ou de certains documents.

Il existe bien évidement plusieurs modèles généraux de qualité [34], [15], [23], [16] définissant les caractéristiques que les auteurs ont jugé nécessaire de formaliser. Ceux-ci mettent en évidence les problèmes rencontrés lors de la conception d'une mesure ou d'un modèle. Ils permettent aussi de formaliser la mesure ou le modèle. Par exemple, certaines mesures mélangent des résultats dont les échelles de valeurs ne sont pas de même type (ex : ordinal, absolu, ratio, etc.). Un autre problème observé et qu'un framework général peut aider à résoudre, est que la mesure utilisée ne donne pas un résultat du même type que celui qu'elle devait fournir. Dans ce cas, la mesure, bien que critiquable, peut toutefois fournir des informations utiles.

1.1.5 Utilisation générale des métriques

Une première façon d'utiliser les modèles et mesures a pour but d'évaluer ce qui n'existe pas encore. C'est le cas pour une grande partie des modèles utilisant comme entrées les documents de conception et de design. Les diagrammes UML vont par exemple servir à estimer la taille de l'application. Parmi ceux-ci, on retrouve les mesures de points de fonction [3] [21] [1] qui font appel à un schéma conceptuel des données et aux cas d'utilisation UML. La deuxième façon d'utiliser les modèles et mesures consiste à vérifier ce qui a déjà été accompli. Le code de l'application est souvent associé à ce type de mesures. Toutefois, l'utilisation du code, des documents de conception, etc., n'est pas restreinte soit à l'évaluation de l'existant, soit à la prédiction sur ce qui est encore à faire. Il existe des modèles faisant appel au code pour la prédiction, comme on le fait pour la maintenance et il existe également des modèles vérifiant le rendement des designers, la clarté d'un document de conception, ou encore la réutilisabilité à partir d'un schéma. Un exemple concret est la mesure du couplage à partir de diagrammes UML.

Il devient donc intéressant de mêler ces deux approches afin d'obtenir plus d'informations. Cette méthode va être appliquée ici puisque nous allons utiliser en même temps un document de conception, le schéma de la base de données, et le code de l'application. Jusqu'à présent, aucun lien fort entre la base de données et l'application n'a été mis en évidence. Nous allons donc tout d'abord évaluer ce que la base de données peut apporter comme informations utiles sur elle-même. Nous étudierons ensuite l'application au travers des informations obtenues sur la base de données.

1.2 A propos de ce mémoire

Avant de poursuivre plus en avant le développement du problème, il est nécessaire de présenter les limites dans lesquelles va s'inscrire ce mémoire. Il se situe dans un contexte exploratoire. En effet, le problème posé dans ce mémoire n'a pas encore été développé ou alors de façon très ponctuelle dans quelques articles. De plus, les développements empiriques que nous présenterons dans ce mémoire, ne permettront pas de vérifier et valider l'ensemble des techniques et outils qui seront développés. Ces phases de validation devront donc être réalisées dans des travaux ultérieurs. Le domaine d'application de ce mémoire se concentre sur la maintenabilité du logiciel. Ce choix permet de limiter le problème et nous permet donc de nous concentrer sur un domaine restreint, mais qui reste encore assez vaste et important pour l'ingénierie logicielle.

Les objectifs de ce mémoire sont de fournir une base théorique et pratique, qui pourra être utilisée dans de futurs développements. Les techniques et outils qui seront développés pourront se baser directement ou par similarité sur des travaux existant. Nous proposerons également des idées neuves. Nous espérons que ces nouvelles techniques et nouveaux outils permettront de progresser dans l'évaluation de la qualité du logiciel et dans l'identification des problèmes de la qualité. Par "évaluation de la qualité du logiciel", nous entendons la mise en place d'outils permettant de quantifier la qualité d'un produit logiciel. L'identification des problèmes correspond à la mise en place d'outils permettant d'identifier les parties du logiciel qui seront probablement difficiles à maintenir. La partie "identification des problèmes" correspond également à la création d'outils qui permettront d'identifier le type de problème.

Ce mémoire débutera par un chapitre d'introduction sur la qualité du logiciel et les méthodes existantes. La qualité du logiciel sera abordée au moyen de modèles de la qualité, reconnus par la communauté, et constituera la première partie du chapitre 2. Un état de l'art sera également présenté dans la deuxième partie du chapitre 2. Il abordera plus en détails les concepts étudiés par les métriques ainsi que certains modèles et mesures. Le

chapitre 3 sera constitué du développement théorique de la maintenabilité vue au travers des bases de données. Nous aurons l'occasion, dans le chapitre 4, d'étudier par la pratique, certains des outils développés au chapitre précédent. Les 2 derniers chapitres regrouperont respectivement les observations et conclusions résultant de la mise en pratique, et les conclusions générales de ce mémoire ainsi que les futures recherches.

Chapitre 2

Qualité du logiciel

Ce chapitre va être organisé en deux parties. La première présentera la qualité du logiciel en général au travers de modèles ISO9126 [17] et de McCall [40]. Dans la deuxième partie, nous présenterons certains modèles et mesures de la maintenabilité.

2.1 Qualité du logiciel : principes généraux

Afin de situer les problèmes qui vont être abordés, l'utilisation d'un modèle de qualité est intéressante. D'une part, un modèle permet d'exprimer des concepts abstraits de manière claire et, d'autre part, certains modèles sont généralement reconnus par l'ensemble de la communauté. Nous présenterons ici deux modèles distincts. Le premier est celui de la norme ISO 9126 (2ème publication) et le deuxième est celui de McCall dont s'est inspiré la norme ISO 9126. Il est important de pouvoir déterminer ce qui va être mesuré car beaucoup d'échecs de modèles et mesures sont dûs à une mauvaise compréhension de ce qu'ils mesuraient exactement. Ces modèles nous serviront donc de ligne de conduite durant le développement de nouvelles méthodes.

2.1.1 Classification ISO et modèle de McCall

La classification donnée par ISO/IEC constitue un standard pour l'évaluation de la qualité logicielle. La première version développée en 1991 à été remaniée afin de préciser certaines caractéristiques des logiciels. La deuxième publication que nous utiliserons a eu lieu en 2001. La norme a été écrite à partir de modèles existant, dont ceux de Boehm et McCall. Dans cette norme, la qualité d'un produit logiciel peut être évaluée en mesurant des attributs internes, externes ou encore en mesurant la qualité lors d'une utilisation du produit. La qualité est décomposée en six caractéristiques, elles-mêmes possédant plusieurs sous-caractéristiques comme le montre le schéma, à la figure 2.1.

Chacune de ces caractéristiques et sous-caractéristiques peut être étudiée sous trois aspects différents :

- Interne ;
- Externe ;
- Lors d'une utilisation.

Nous ne verrons ici que les aspects internes étant donné les moyens nécessaires à une évaluation externe. Les valeurs des caractéristiques externes et en utilisation dépendent en

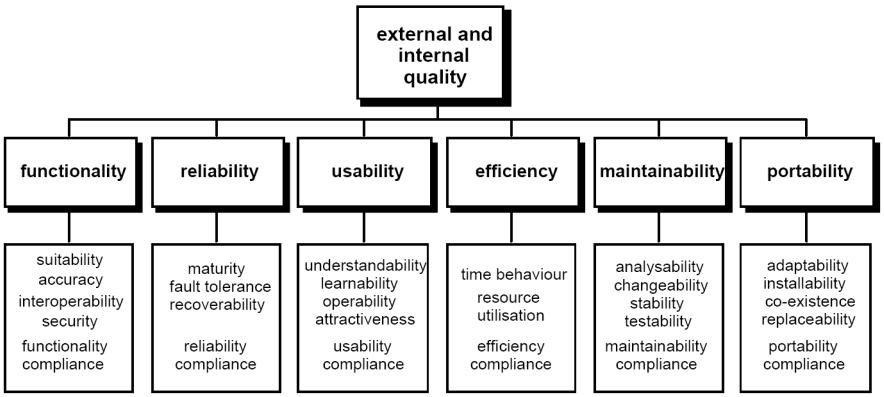

FIG. 2.1 – Modèle de la qualité du logiciel, ISO/IEC 9126 [17]

effet de facteurs extérieurs au logiciel. Ces facteurs sont, par exemple, les compétences de la personne chargée du processus de maintenance ou encore les compétences d'un utilisateur face au système. Les aspects internes ne dépendent, quant à eux, que de facteurs internes au logiciel. La différence entre interne et externe peut sembler ambigüe. L'exemple suivant précise cette différence dans le cas de la maintenabilité. Dans le cadre d'un processus de maintenance, le nombre et le type d'erreurs restant dans le logiciel est un élément interne de la maintenabilité. En effet, quel que soit l'environnement, ce nombre ne change pas. Le temps nécessaire à la correction de ces erreurs dépend, en revanche, des capacités de l'analyste et de l'erreur elle-même. Les caractéristiques externes peuvent également servir à valider des hypothèses faites au sujet l'influence des caractéristiques internes sur les caractéristiques externes. Dans des environnements externes équivalents, les valeurs des caractéristiques ou sous-caractéristiques externes permettent, en effet, d'étudier l'influence de l'aspect interne. Cette validation empirique est nécessaire et doit, pour certains modèles, être réalisée pour chaque entreprise afin de calibrer des paramètres du modèle ou de la mesure.

En plus de caractéristiques, la norme définit également le concept d'attribut du logiciel. Un attribut représente une propriété d'un produit logiciel. Les attributs peuvent être soit internes, soit externes et il existe une relation entre les attributs internes et externes. Les attributs internes et externes permettent respectivement de quantifier une ou plusieurs qualités internes et externes du logiciel. La norme présente des mesures purement internes qui sont utilisées pour mesurer certains attributs du design du logiciel ou du code. Le résultat de ces mesures permet d'évaluer en totalité ou en partie la valeur d'une ou plusieurs caractéristiques et sous-caractéristiques du logiciel. Ces mesures correspondent à la définition de mesure que nous avons vue dans la section 1.1.1.

La maintenabilité étant le point central de ce mémoire, voici sa définition dans la norme ISO/IEC 9126 :

"The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of software to changes in environment, and in requirements and functional specifications."

La maintenabilité d'un produit logiciel correspond à sa capacité à être modifié. Les modifications peuvent inclure des corrections, des améliorations ou une adaptation du logiciel à des changements dans l'environnement, dans les exigences et dans les spécifications fonctionnelles. Parmi les cinq sous-caractéristiques de la maintenabilité proposées dans

le modèle de la norme ISO, seules deux ont été reprises. Celles-ci sont l'analysibilité et l'évolutivité.

- Analysibilité : *"The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified."* L'analysibilité correspond donc à la facilité avec laquelle il est possible d'identifier les déficiences et les causes d'erreurs dans le produit logiciel ainsi que la facilité avec laquelle on peut identifier les parties qui doivent être modifiées.
- Évolutivité : *"The capacity of the software product to enable a specified modification to be implemented."* Le terme anglais utilisé dans la norme est " changeability ", que nous avons traduit ici par " évolutivité ". L'évolutivité est donc la capacité du logiciel à permettre à une certaine modification d'être implémentée.

Comme nous l'avons dit précédemment, il existe d'autres sous-caractéristiques de la maintenabilité. Celles-ci sont la stabilité, la testabilité et la maintenabilité générale. La maintenabilité générale représente la maintenabilité moyenne calculée à partir des autres sous-caractéristiques de la maintenabilité. La testabilité est la capacité du produit logiciel à être validé, testé après un changement du système (*"The capability of the software product to enable modified software to be validated."* [17]). La stabilité est la capacité du produit logiciel à éviter des effets inattendus suite à une modification du produit logiciel. On constate que les définitions se trouvant dans la norme ISO restent assez vagues contrairement à d'autres modèles. Bien qu'une marche à suivre pour l'utilisation des modèles et mesures soit définie, la norme aborde assez peu la façon dont ils peuvent être utilisés pour mesurer les attributs et caractéristiques du logiciel.

En ce qui concerne le choix des sous-caractéristiques, nous avons retenu celles qui intuitivement semblent avoir le plus de relation avec les objectifs décrits à la fin du chapitre 1.

Le modèle suivant, celui de McCall, à la figure 2.2, possède une structure semblable à celui se trouvant dans la norme. Toutefois, les caractéristiques de la qualité logicielle sont divisées en sous-caractéristiques qui se rapprochent plus des attributs internes du logiciel. Ce modèle est construit sur base de l'utilisation faite du produit final (une utilisation, une révision ou une transition du produit). Voici la partie du modèle concernant la maintenabilité. L'ensemble du modèle est présenté dans l'ouvrage de McCall [40].

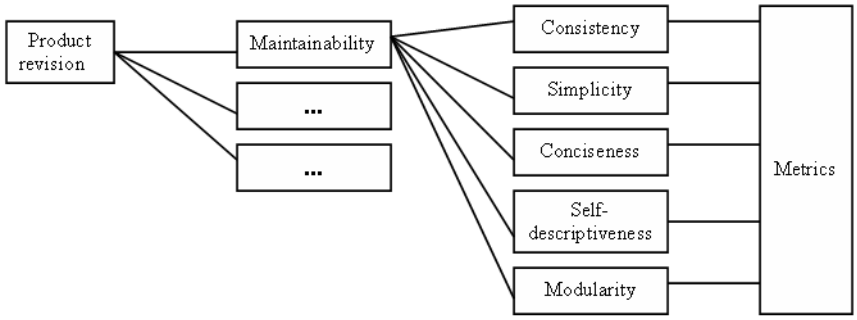

FIG. 2.2 – Modèle de la qualité du logiciel de McCall [40]

Les éléments de ce modèle sont différents de ceux de la norme puisque McCall parle d'utilisation faite du produit pour l'élément le plus à gauche, c'est-à-dire la révision d'un produit. La maintenabilité est un facteur et non plus une caractéristique. Finalement, on parlera de critères pour la cohérence, la simplicité,etc. L'intérêt de ce modèle réside dans le fait que McCall y a présenté explicitement les attributs internes, ou du moins les classes d'attributs internes du logiciel. Ces attributs internes influencent la caractéristique étudiée.

2.1.2 La maintenabilité, ses sous-caractéristiques et leurs attributs

Nous avons défini la maintenabilité ainsi que les sous-caractéristiques que nous allons étudier. Cette section présente les attributs qui vont être utilisés pour l'analyse de la maintenabilité. Il ne s'agit ici que d'une présentation brève des attributs internes. Ceux-ci seront détaillés dans le chapitre 3 consacré aux nouvelles techniques et outils ainsi qu'à la section 2.2.

Le premier attribut est la taille. La taille d'un produit logiciel n'est généralement pas utilisée comme un attribut interne intervenant dans les modèles de qualité. La taille permet néanmoins de calculer les coûts du logiciel et revêt donc une grande importance. La taille va être abordée ici de deux manières différentes. Tout d'abord en termes de lignes de code et ensuite en termes de fonctions.

Nous verrons ensuite la complexité, car son influence sur la maintenabilité est reconnue. Il existe également plusieurs types de complexité. La complexité structurelle du code est étudiée au travers d'un graphe représentant le code d'une fonction ou d'un module de l'application. L'autre complexité abordée par la suite, sera la complexité "cognitive", qui s'approche le plus de ce qu'on entend généralement par complexité. Elle regroupe la notion de structure, mais également la notion de sens. Elle étudie la difficulté avec laquelle un produit logiciel peut être compris. La complexité est aussi liée à la simplicité et la concision , qui influencent la maintenabilité d'un système [40].

Enfin, la cohésion est un attribut souvent utilisé lors d'une estimation de la qualité d'un programme écrit en orienté objet mais elle trouve aussi son utilité dans d'autres langages et dans d'autres contextes. Par exemple, la cohésion d'un logiciel peut être centrée sur les données et leur utilisation.

2.1.3 Utilité d'une telle classification

Nous avons choisi d'utiliser ces deux modèles afin de pouvoir définir de nouvelles méthodes qui, d'une part, se rapprochent plus de la norme ISO/IEC et, d'autre part, restent assez pratiques dans l'évaluation de la qualité du logiciel. Ces deux modèles restent assez proches. En effet, bien que présenté en 1977, le modèle de McCall contient les mêmes caractéristiques que la norme ISO 9126 et permet de définir plus précisément les attributs à mesurer pour estimer les caractéristiques. La norme ISO, quant à elle, formalise les aspects internes et externes de la qualité d'un logiciel, ainsi que de nouvelles sous-caractéristiques. La norme ISO permet donc de rester à un niveau d'abstraction assez élevé et offre certaines libertés quant à la mise en place du modèle.

2.2 Applications à la maintenabilité

Avant de commencer le développement de nouvelles techniques et méthodes, il est important d'analyser les travaux existant. Cette analyse va mettre en évidence les techniques utilisant la même approche que celle décrite dans ce mémoire, c'est-à-dire l'évaluation de la qualité à partir des bases de données et des accès aux données. Elle va également présenter des techniques ayant leur utilité dans les processus de maintenance dont les concepts peuvent être adaptés aux bases de données. Evidemment, une adaptation des modèles déjà validés ne permettra pas de valider automatiquement les nouveaux modèles dérivés. En effet, les validations empiriques réalisées pour les modèles existant ne seront plus utilisables étant donné les changements de domaines, d'objets, etc., constituant le nouveau modèle. Les modèles développés par la suite demanderont donc de nouvelles études empiriques afin d'être validés.

2.2.1 Modèles de la norme ISO/IEC 9126

La norme ISO/IEC 9126 propose un ensemble de modèles liés à la maintenabilité. La classification de ceux-ci se fait sur base du type de modèles, c'est-à-dire interne ou externe ainsi que sur la sous-caractéristique concernée. Dans cette section, nous donnerons les modèles les plus intéressants, qui peuvent avoir une relation avec les principes d'analyse de la qualité développés dans ce mémoire. Les points suivants sont de type externe et font donc intervenir des éléments extérieurs au logiciel.

- L'analysibilité et l'étude des capacités de diagnostic : il existe plusieurs façons d'estimer cette capacité. Seuls deux modèles ont été retenus. Ceux-ci concernent le diagnostic d'erreurs, c'est à dire l'identification et la compréhension d'une erreur. Le premier modèle est :

  $$X = A/B$$

  où A est le nombre d'erreurs que l'analyste a pu détecter et B est le nombre total d'erreurs. Le second donne la capacité à analyser l'erreur, dont la formule est :

  $$X = 1 - A/B$$

  où A est le nombre d'erreurs dont les causes ne sont toujours pas comprises et B le nombre total d'erreurs déjà trouvées. Afin d'améliorer ces résultats, un processus d'aide à l'identification des erreurs est intéressant à mettre en place. Ce processus aurait pour but d'identifier les structures complexes, induisant en erreur, ou encore les modules dont les activités qu'ils prennent en charge sont trop différentes, etc.
- L'évolutivité et la complexité : un seul modèle est repris ici. Il s'agit d'établir la complexité d'une modification faite par une personne chargée de la maintenance. La formule utilisée est la suivante :

  $$T = Sum(A/B)/N$$

  où A est le temps passé sur le changement, B est la taille du changement et N est le nombre de changements. Bien que dépendant fortement des compétences du mainteneur, cette formule fait également intervenir des attributs internes de l'élément à changer. Le plus évident est la taille, mais la complexité du code ou du schéma, la cohésion, le couplage, etc., influencent également la difficulté de la tâche. Toutes les informations concernant les problèmes identifiés facilitent donc la tâche de l'analyste.

Les modèles évaluant des caractéristiques externes font intervenir des paramètres qui ne peuvent être estimés en analysant les données ou le code. Pour cette raison, nous mettrons principalement en place par la suite des outils de diagnostic utilisant des attributs internes. Ces outils auront pour but d'identifier les problèmes possibles du logiciel en terme de maintenabilité et s'appuieront sur une évaluation des attributs internes au produit mesuré.

Les modèles qui viennent d'être présentés nous ont semblé être les plus en relation avec les attributs internes du logiciel. Il existe bien entendu d'autres modèles que nous n'avons pas souhaités reprendre ici.

2.2.2 Concepts et notions caractérisant le logiciel

Cette section détaille quelques attributs, modèles et mesures existant dans l'ensemble de l'ingénierie du logiciel et pouvant être adaptés aux bases de données et à leurs accès.

Les attributs internes au logiciel sont les éléments les plus souvent utilisés lors d'une évaluation de la qualité d'une application. Cela est dû à leur aspect concret et à l'existence de nombreuses méthodes leur garantissant une utilisation pratique. Toutefois ces attributs ne permettent pas à eux seuls de calculer, par exemple, l'analysibilité d'un système

étant donné l'intervention de caractéristiques externes. Malgré cela, de nombreux travaux ont étudié et quantifié la relation existant entre ces attributs et les caractéristiques du logiciel. D'autre part, ces attributs interviennent également dans les diagnostics de qualité. L'exemple le plus probant est la complexité de McCabe[39] qui permet d'identifier les structures complexes dans le code du logiciel. On remarque que cette mesure intervient également dans certains modèles quantifiant la maintenabilité. Cet état de l'art va donc présenter des modèles et mesures des attributs généralement reconnus.

La taille

La taille d'un logiciel peut être calculée en lignes de code et en points de fonctions[7]. Les lignes de code ont constitué la première mesure couramment utilisée de la taille d'un programme. Les lignes de code servent généralement à estimer les coûts ou un rendement. Leur calcul est simple et automatisable, ce qui les rend attrayantes. Il existe plusieurs méthodes pour le calcul du nombre de lignes de code. En effet, certaines méthodes considèrent l'ensemble du fichier source, ce qui tient compte des commentaires et des lignes vides. En supprimant ces deux types de lignes, le résultat de la mesure donne exactement le nombre de lignes contenant du code. Mais une instruction peut se trouver sur plusieurs lignes et une même ligne peut contenir plusieurs instructions. Il existe donc une mesure de la taille du code, reprise dans les mesures du nombre de lignes de code, qui compte le nombre d'instructions contenues dans le code. L'échelle de mesure a donc changé en passant du nombre de lignes de code au nombre d'instructions. Ces mesures possèdent des caractéristiques que nous exploiterons par la suite. La première est que toutes ces mesures restent techniques et il est donc simple de mettre en place un algorithme effectuant ce comptage. La deuxième caractéristique commune, qui a une relation étroite avec la première, est que ces mesures s'appuient sur la syntaxe du code, et donc des éléments "physiques" du code. C'est dans cette approche que de nombreux modèles comptent les éléments d'un artefact autre que le code du logiciel. Par exemple, la taille d'un diagramme de classes UML [38] est évaluée sur base du nombre de classes, d'attributs et de méthodes. L'article présente également un indice de complexité pour ces diagrammes. Celui-ci est évalué en fonction du nombre total d'associations et du nombre d'association d'un certain type (agrégation, dépendance, généralisation, etc.). Nous réutiliserons par la suite ce type de mesure pour l'analyse des structures de données représentées sous formes de schémas conceptuels et logiques.

Contrairement au nombre de lignes de code, la taille fonctionnelle analyse les fonctions proposées par le logiciel. Cette taille a été développée par Albrecht [3], puis a été dérivée en plusieurs modèles et mesures de taille fonctionnelle. L'analyse de documents ou du code permet d'estimer le nombre de fonctions du programme ainsi que leur taille respective. Il est ensuite possible, en tenant compte de certains paramètres, de dériver une estimation des SLOC[1] du logiciel. Cette mesure est généralement utilisée pendant l'analyse des besoins au travers des cas d'utilisation et du diagramme des données. Il n'existe pas de moyen complet pour automatiser cette mesure, étant donné l'importance du contexte sémantique et pragmatique dans ces documents. Une autre approche à cette mesure consiste à dériver la taille fonctionnelle par une rétro-ingénierie du code. Cette méthode a son utilité lorsque la taille fonctionnelle est inconnue et qu'elle est nécessaire pour un nouveau processus et que la documentation n'est pas suffisante. Cette approche, bien que plus rigoureuse, n'est pas encore automatisée malgré plusieurs tentatives. Par la suite, nous verrons comment les bases de données et leurs accès, peuvent permettre d'estimer, sous certaines conditions, la taille fonctionnelle.

[1] Source Line Of Code.

La complexité

Le terme "complexité" peut être interprété de plusieurs manières. La première signification concerne les théories de la calculabilité et de la complexité, on parle alors de la complexité du problème et de la solution. La complexité désigne également la difficulté avec laquelle une personne parvient à comprendre le code d'un programme, un schéma, etc. Dans le domaine des métriques, la complexité désigne également la mesure structurelle du code. Cette mesure se base sur une représentation du code sous forme de graphe de contrôle de flux. Plus ce graphe est complexe, plus le code est considéré comme étant complexe. Nous parlerons, dans ce cas, de complexité structurelle. Nous ne considèrerons ici que les deux dernières significations de la complexité.

La complexité structurelle est certainement la plus connue, en particulier au travers de la complexité cyclomatique de McCabe [39]. Il est possible de représenter le code d'un logiciel sous la forme d'un graphe orienté. A partir de ce graphe, on peut analyser les structures du code. On peut définir des patterns de bonnes structures qui doivent composer le code du module analysé. A partir de ces informations, on peut alors compter le nombre d'arcs, de noeuds, d'occurrences d'une structure de base ou encore vérifier quelle partie du code ne respecte pas un pattern reconnu.

La complexité cyclomatique de McCabe compte le nombre de noeuds de décision d'une fonction ou d'une méthode et ajoute 1. Cette mesure est sans doute la plus utilisée des mesures de complexité structurelle, même si le résultat ne prend en compte qu'un aspect limité de la structure du code. La valeur maximum conseillée par l'auteur de la mesure est de 10 pour une fonction ou méthode. En plus d'être un outil de diagnostic, elle intervient, comme nous l'avons cité précédemment dans des modèles évaluant les caractéristiques de la qualité. Par exemple, le modèle suivant [15] [14] :

$$R = 0.295a - 0.499b + 0.13c$$

où R est la lisibilité, a est la taille moyenne normalisée des variables en nombres de caractères, b est le nombre de lignes contenant des instructions et c est la complexité cyclomatique de McCabe [14].

La complexité structurelle ne semble pas avoir été appliquée aux accès réalisés sur les bases de données. Il n'existe en effet pas de mesure de complexité spécifique aux accès SQL. Pourtant, l'implémentation d'une requête demande un effort de la part du programmeur. Un effort est également nécessaire lors de la maintenance. Nous étudierons, dans la partie 3.4.1, la complexité des requêtes.

Nous étudierons également dans la section 3.3, les caractéristiques des schémas conceptuels, dont la complexité.

La cohésion

La cohésion du logiciel est définie dans [15] de la façon suivante :

" The cohesion of a module is the extent to which its individual components are needed to perform the same task."

La cohésion étudie donc la façon dont les fonctions, méthodes, classes, etc., d'un même module effectuent des tâches de même nature. Ces tâches peuvent être semblables par le fait qu'elles utilisent un même groupe de données, qu'elles effectuent un même type de traitement, qu'elles fournissent un même type de résultat, etc.

Une échelle de cohésion a été définie en 1979 par Yourdon et Constantine [47] afin de pouvoir placer les modules d'un logiciel selon leur cohésion dans une échelle ordinale. Cette échelle contient 7 catégories de cohésion et est définie comme suit :

- Cohésion coïncidente : les parties d'un module sont groupées arbitrairement et n'ont pas de relation entre elles. Le module regroupe alors des fonctions n'ayant aucun rapport entre elles ;
- Cohésion logique : les parties du module sont regroupées sur base du traitement qu'elles effectuent. Le module effectue alors plusieurs fonctions liées logiquement ;
- Cohésion temporelle : les parties du modules sont regroupées en fonction du moment où elles sont utilisées dans le programme. Les fonctions du module sont donc exécutées dans un même espace de temps ;
- Cohésion procédurale : les parties du module sont regroupée car elles suivent toujours une certaine séquence d'exécution. Le module contient plus d'une fonction et ces fonctions sont liées à une procédure plus générale du logiciel ;
- Cohésion communicationnelle : les parties d'un module sont regroupées en fonction des données sur lesquelles elles travaillent. Le module possède plusieurs fonctions mais celles-ci s'effectuent sur le même groupe de données ;
- Cohésion séquentielle : les parties du module s'enchaînent de façon déterminée et les outputs d'une partie sont les inputs d'une autre. Les fonctions s'enchaînent donc dans le module en suivant l'ordre décrit dans les spécifications du module ;
- Cohésion fonctionnelle : les parties du module contribuent à l'accomplissement d'une seule tâche. Le module n'est alors défini que pour accomplir une seule fonction.

Cette échelle ordinale permet donc de classifier les modules en fonction de leur cohésion. La cohésion la plus mauvaise étant la cohésion coïncidente et la meilleure cohésion étant la fonctionnelle.

Malheureusement, cette échelle ne constitue qu'une mesure de la cohésion d'un module et ne permet pas de comparer des modules se trouvant au même niveau de l'échelle. Elle ne permet pas non plus de définir la cohésion générale de l'application. Il est toutefois possible d'utiliser les indices statistiques et des ratios afin d'établir la cohésion générale du système.

Plusieurs méthodes ont également été développées sur base des travaux de Yourdon et Constantine. Parmi celles-ci, se trouvent les mesures développées par Bieman et Ott [5]. La méthode utilisée par Bieman et Ott se focalise sur les outputs des fonctions et des méthodes du programme. Ils définissent trois modèles de cohésion basés sur un slicing des données et sur une étude des relations entre les variables. Ces trois modèles définis évaluent la " cohésion fonctionnelle forte ", la " cohésion fonctionnelle faible " et l' " adhérence ". Le premier de ces modèles permet de relier la fonction avec les meilleures cohésions décrites dans l'échelle de Yourdon et Constantine. Le deuxième relie la fonction avec les cohésions les plus faibles de l'échelle. Le troisième donne quant à lui un indice statistique des apparitions groupées des variables par rapport au nombre d'instructions de la fonction.

Afin d'illustrer la méthode développée par Bieman et Ott, nous allons commencer par définir brièvement ce qu'est le slicing. Le slice d'un module à l'instruction i par rapport à la variable v est l'ensemble de toutes les instructions et de tous les prédicats du module qui peuvent influencer la valeur de v à i [46].

Dans la méthode développée par Bieman et Ott, on commence par identifier les outputs de la fonction choisie. Cette identification est suivie par un slicing sur chacun de ces outputs. Le résultat du slicing est donc, pour chaque output, l'ensemble des instructions et prédicats qui ont affecté l'output dans la fonction. Dans cet ensemble, on peut identifier les variables et valeurs se trouvant dans les instructions et prédicats. Une même variable ou valeur peut survenir plusieurs fois dans la fonction et donc dans le slice. De ce fait, on identifiera l'occurence des variables et des valeurs par le nom de la variable ou de la valeur et son numéro d'apparition dans la fonction. La figure 2.3 met en évidence les variables et valeurs se trouvant dans le slice de l'output $SumN$.

Une fois ces ensembles d'occurences de variables et de valeurs identifiés, la méthode

```
procedure SumAndProduct
  ( N : integer;
    var SumN, ProdN : integer );
var
   I : integer;
begin
   SumN := 0;
   ProdN := 1;
   for I := 1 to N do begin
      SumN := SumN + I;
      ProdN := ProdN * I
   end
end;
```

FIG. 2.3 – Illustration du slicing sur la variable SumN [5]

met en relation les slices des outputs. On peut alors voir quelles occurences de variables et de valeurs se trouvent dans plusieurs slices ou dans un seul. La figure 2.4 illustre ces relations.

Data Token	SumN	ProdN
N_1	|	|
$SumN_1$	|	
$ProdN_1$		|
I_1	|	|
$SumN_2$	|	
0_1	|	
$ProdN_2$		|
1_1		|
I_2	|	|
1_2	|	|
N_2	|	|
$SumN_3$	|	
$SumN_4$	|	
I_3	|	
$ProdN_3$		|
$ProdN_4$		|
I_4		|

FIG. 2.4 – Abstractions des slices[5]

Un comptage des outputs influencés par la même occurence de variable ou de valeur de la fonction est alors effectué. Les trois modèles introduits précédemment peuvent alors être appliqués sur base des résultats de ce comptage. Les échelles de ces modèles n'ont pas été prouvées comme étant des échelles de ratio mais leurs valeurs se situent entre 0.0 et 1.0, ce qui permet les comparaisons entre différentes fonctions du logiciel. Plus de détails concernant ces méthodes sont disponibles dans [41] et [5].

Il est important de voir que le slicing a permis d'évaluer une caractéristique subjective du logiciel. La méthode développée par Bieman et Ott est automatisable et permet de situer le résultat par rapport à l'échelle de Yourdon et Constantine.

En ce qui concerne la cohésion, la cohésion communicationnelle semble avoir un lien fort avec les bases de données. En effet, selon la définition que nous avons donné

précédemment, cette cohésion se base sur les données utilisées par les fonctions du module. Il peut donc être intéressant d'étudier la cohésion sur base des données obtenues par la fonction au moyen d'une requête à la base de données. Cette étude nous permettrait d'affirmer si en termes de données, un module composé de plusieurs fonctions est cohérent ou ne l'est pas. Il peut également être intéressant d'ajouter à la méthode proposée par Bieman et Ott, une prise en compte des accès aux bases de données. Il existe en effet des techniques permettant de suivre une même variable présente à la fois dans le code procédural et dans le code SQL de l'application [12]. Il ne s'agirait donc pas d'une modification fondamentale de la méthode. Cette modification étendrait le champ d'application de la méthode. Les outils concernant la cohésion seront présentés dans la partie 3.5.1.

Caractéristiques non vues

Il existe d'autres caractéristiques liées à la maintenabilité que nous n'aborderons pas ici.[2] Une d'entre-elles est le couplage [15] qui peut être étudié en suivant des règles très précises. Malheureusement, celui-ci s'applique principalement aux appels de fonctions entre les modules. Ces appels ne sont pas présents au niveau d'un schéma de données et sont peu utilisés au niveau du code SQL. Il est vrai que certaines procédures SQL peuvent être appelées par d'autres ou encore par du code JAVA ou COBOL mais elles n'appellent généralement pas d'autres fonctions et ne constituent pas un module à elles seules.

Deux autres caractéristiques sont la lisibilité et la concision. Pour ces deux caractéristiques, nous donnerons seulement quelques observations. Mais nous ne les étudierons pas en détails dans ce mémoire car elles se basent sur des éléments subjectifs pour leur évaluation.

2.2.3 Travaux et recherches abordant le problème au travers des bases de données

Les travaux abordant directement la qualité du logiciel par les bases de données sont peu nombreux. Nous donnerons ici deux exemples. Le premier se rapporte à la qualité et l'analyse du langage SQL et plus précisément des requêtes faites à la base de données. Dans l'article [45], les auteurs abordent les problèmes de reconnaissance des requêtes dans le code du logiciel et de l'analyse de ces requêtes par des moyens statistiques, comme le comptage du nombre de variables, de sorties et d'entrées, du nombre de requêtes ayant une même structure, du niveau moyen d'imbriquation des requêtes, etc. Les auteurs proposent également, au travers d'une étude de cas, des conclusions sur la qualité du logiciel en fonction des résultats obtenus lors de l'analyse. Ils identifient, entre autres, grâce aux résultats de leurs mesures, des requêtes dupliquées, des erreurs de structure dans la base de données et des constructions SQL mal définies. Cet article propose donc une application très concrète de l'analyse des requêtes et développe des outils pour la maintenabilité du logiciel.

Le deuxième article [43] traîte de l'estimation de la taille d'un système d'informations à partir du schéma conceptuel des données. Les auteurs ont utilisé les analyses statistiques et la régression linéaire pour obtenir une formule qui, sur base du nombre de types d'entités, de types d'associations, d'attributs et d'attributs par type d'entités, permet d'estimer la taille en lignes de code (SLOC) du système. Selon le type de système, ils donnent des coefficients multiplicateurs aux quatres éléments que nous venons de citer. La formule utilisée est la suivante :

$$KLOC = \widehat{\beta}_0 + \widehat{\beta}_C C + \widehat{\beta}_R R + \widehat{\beta}_{\overline{A}} \overline{A}$$

[2] Une liste d'autres mesures et modèles est disponible dans [19].

où l'unité de $KLOC$ est en milliers de lignes de code, les β représentent les coefficients à estimer, C est le nombre de types d'entités, R est le nombre de types d'associations et $\overline{A}$ est le nombre moyen d'attributs par type d'entités, obtenu au moyen de A, du nombre total d'attributs et de C.

Les coefficients β varient selon le type du logiciel. Les types de logiciels présentés dans l'étude de cas sont, par exemple, les systèmes professionnels écrits en Visual Basic, les systèmes Open Source développés en PHP, les logiciels écrits en JAVA dans l'industrie, etc. Ces coefficients sont présentés dans l'article et sont justifiés par les résultats de nombreuses études de cas.

L'intérêt de cette méthode est de pouvoir estimer la taille du logiciel avant son implémentation. Elle n'est donc pas réellement utile dans le cadre de la maintenabilité, puisqu'on connaît en principe le nombre de lignes de code lors d'une phase de maintenance. Cette méthode prouve néanmoins qu'il existe un lien direct entre la taille du logiciel et la structure de données. Nous verrons par la suite qu'il est possible d'estimer la taille d'un système d'informations, en points de fonctions, en utilisant la base de données et les accès SQL.

Chapitre 3

Techniques et outils

Les chapitres précédents ont introduit les concepts de qualité du logiciel, de mesures et de modèles des logiciels. Ce chapitre est consacré à la définition et au développement de nouveaux outils. Ceux-ci auront pour but d'évaluer les caractéristiques de la structure des données et des accès à celles-ci. Ils permettront également de fournir un diagnostic quant à certains aspects de la maintenabilité du logiciel. Dans le cas d'une relation évidente entre, d'une part, les données et les accès à celles-ci et, d'autre part, la maintenabilité, cette nouvelle approche permettra d'améliorer les capacités d'analyse de la maintenabilité et donc de diminuer les risques lors d'un processus de maintenance.

Ce chapitre commencera par une description des ressources, c'est-à-dire des éléments qui pourront être analysés par les techniques et outils que nous développerons par la suite. La partie suivante présentera de façon plus détaillée les intuitions de départ qui sont à la base de ce mémoire et l'approche utilisée pour les développements qui suivront. Nous développerons ensuite les nouvelles méthodes et mesures, en commençant par une analyse du schéma des données, pour ensuite se tourner vers les accès aux données et pour finir sur l'influence de ces deux derniers éléments sur le logiciel.

3.1 Ressources utilisées lors des analyses

Avant de continuer plus en avant dans le développement de nouveaux modèles et métriques, il est important de présenter le cadre "technique" des outils et modèles qui vont être développés ici. Par cadre technique, nous entendons les ressources qui vont être utilisées en vue d'obtenir de nouvelles informations sur le logiciel. Ces ressources sont de deux types. Le premier type de ressources est une représentation des données, que ce soit sous la forme d'un schéma conceptuel ou logique. Le deuxième type de ressources regroupe le code exécuté lors de l'utilisation du logiciel. Ce code comprend le code procédural de l'application ainsi que le code SQL permettant l'accès, l'insertion, la mise-à-jour et la suppression des données.

3.1.1 Schéma de données

Schéma conceptuel

Les informations suivantes, sur le modèle entité-association, sont tirées de [26].

Le schéma conceptuel permet de représenter les différents objets d'un domaine d'application ainsi que les attributs de ces objets et les relations qu'ils ont entre eux. Parmi les

différents types de schéma conceptuel, nous présenterons le modèle entité-association, appelé également modèle entité-relation. Nous utiliserons ici une version étendue du modèle original [10]. Cette version étendue est celle utilisée par l'outil DB-MAIN. Par la suite, nous abrègerons le nom du modèle par modèle ERA (entité-relation-association).

Le modèle ERA permet de modéliser les objets du domaine d'application sous forme de type d'entités. Un type d'entités représente la catégorie sémantique à laquelle appartiennent certains objets du domaine d'application. Par exemple, le type d'entités **Client** représente l'ensemble des clients du domaine d'application. Afin de représenter les relations existant entre les entités, nous utilisons les types d'associations. Un type d'associations désigne une relation définie sur une collection de type d'entités. Un type d'associations permet donc de représenter une relation existant entre les objets du domaine d'application, ces objets étant représentés, dans le modèle, par les types d'entités concernés par l'association. Ces deux éléments forment les constructions principales du modèle ERA.

En ce qui concerne les types d'entités, en plus du nom de la classe d'objets qu'ils représentent, les types d'entités possèdent des attributs. Les attributs d'un type d'entités sont les propriétés communes à toutes les entités de ce type. Par exemple, le nom des clients du domaine d'application, sera représenté par un attribut **nom** dans le type d'entités **Client**. Les types d'associations peuvent également posséder des attributs. Comme pour les types d'entités, les attributs d'un type d'associations désigne une propriété commune à toutes les associations de ce type. Afin d'illustrer la théorie, nous utiliserons le schéma à la figure 3.1. Dans ce schéma, les types d'entités sont les clients, les produits, les commandes et les détails. On constate qu'un client possède un numéro de client, un nom, une adresse, etc. Ces caractéristiques sont les attributs du type d'entités **Client**. Dans le schéma, les types d'associations sont **passe**, **de** et **en**.

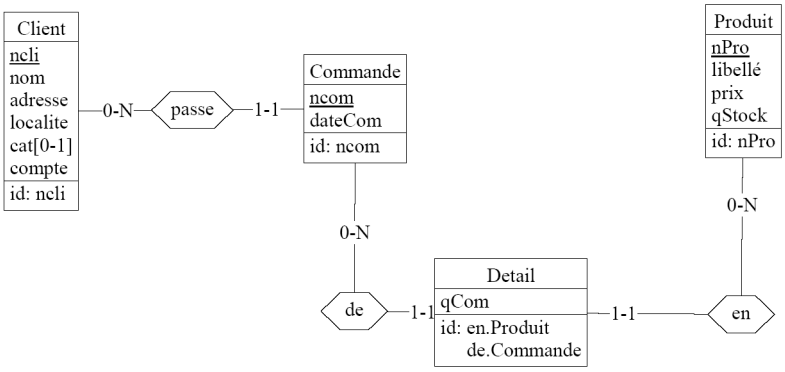

FIG. 3.1 – Exemple de schéma ERA

Les attributs possèdent plusieurs caractéristiques. Ils peuvent être atomiques ou composés. Un attribut atomique ne peut être fragmenté en d'autres composants significatifs. Au contraire, les attributs composés sont constitués d'éléments significatifs par rapport au domaine d'application. Les éléments composant un attribut sont également des attributs et on dit alors qu'un attribut composé est le parent des attributs qui le composent. Les attributs qui ne composent pas un attribut ont pour parent soit le type d'entités, soit le type d'associations auquel ils appartiennent.

Les attributs possèdent une cardinalité, représentée sous la forme **[I-J]**, où **I** et **J** sont des entiers positifs, **J** est strictement supérieur à **0** et **I** est inférieur ou égal à **J**. **I** indique le nombre minimum d'occurences de l'attribut pour chaque occurence de son parent et **I** est supérieur ou égal à **0**. **J** indique quant à lui le nombre maximum d'occurences. Suivant sa cardinalité, l'attribut possède les caractéristiques suivantes. Il est soit obligatoire si **I**

$\geq$ **1** soit facultatif si **I = 0**. L'attribut est également soit monovalué, soit multivalué. Il est monovalué lorsque **J = 1** et il est multivalué lorsque **J** $>$ **1**. Généralement, les attributs sont obligatoires et monovalués. Les attributs ayant ces deux caractéristiques sont généralement représentés graphiquement sans leur cardinalité dans le modèle ERA étendu. Enfin, **J** peut être égal à **N**, signifiant qu'il y a plusieurs occurences dont le nombre est indéterminé.

La dernière caractéristique des attributs concerne la stabilité de la valeur de l'attribut. L'attribut est dit stable s'il ne peut être modifié après assignation et est dit modifiable dans le cas contraire. Dans le schéma à la figure 3.1, tous les attributs sont atomiques, obligatoires, monovalués et modifiables, à l'exception de l'attributs **cat**, du type d'entités **client** qui est atomique, facultatif, monovalué et modifiable.

Nous allons maintenant développer les types d'associations. Les types d'associations ont un nom (le nom de la relation qu'ils représentent) et peuvent également posséder des attributs. Chaque type d'associations possède aussi un degré qui correspond au nombre de types d'entités concernées par la relation représentée par le type d'association. S'il y a deux types d'entités, le degré est dit binaire. S'il y en a trois, le degré est ternaire. S'il y en a n, le degré est dit n-aire. Nous mettons en évidence le fait que lorsqu'un type d'entités est présent plusieurs fois dans la relation, chaque occurence de ce type d'entités est considérée. Par exemple, si un type d'entités a une relation avec lui-même, le degré sera binaire. Un type d'associations est généralement de degré binaire ou ternaire, il est rare d'en observer de degré supérieur. A chaque occurence d'un type d'entités dans la relation, on associe un rôle. Ce rôle possède un nom et une cardinalité. Le nom précise le rôle du type d'entités dans la relation. La cardinalité du rôle est de type **[I-J]**, où **I** indique le nombre minimum d'association dans lesquelles une entité peut jouer ce rôle et **J** indique le nombre maximum. Dans l'exemple, tous les types d'associations sont binaires et représentent une relation de type 1 à N (un-à-plusieurs). Pour la relation **passe**, une commande est toujours liée à au plus un client. Dans l'autre sens, un client peut avoir plusieurs commandes. Il existe deux autres types de relation binaire. Ils sont la relation 1 à 1 (un-à-un) et la relation N à N (plusieurs-à-plusieurs). Les nombres 1 et N font référence au maximum de la cardinalité des rôles.

Les derniers éléments que nous allons présenter sont les identifiants et les contraintes d'intégrité. Un identifiant d'un type d'entités représente une caractéristique unique d'une entité par rapport aux entités de même type. L'identifiant peut être un attribut ou un ensemble d'attributs et ou de rôles attachés au type d'entités. Chaque valeur d'un identifiant d'un type d'entités étant unique, elle permet de retrouver l'entité qui lui est associée. Il existe donc une contrainte d'unicité sur les valeurs des identifiants.

Les contraintes d'intégrité sont du point de vue du domaine d'application, "les propriétés des objets du domaine d'application qui limitent les configurations et les comportements" [26]. En base de données, elles sont vues comme "les propriétés formelles que les données doivent respecter à tout instant, ou à des instants déterminés" [26]. Une contrainte d'intégrité correspond généralement à la traduction d'une contrainte du domaine d'application. Il existe de nombreuses contraintes différentes. Par exemple, la contrainte d'unicité sur la valeur d'un identifiant ou encore les cardinalités des rôles sont des contraintes d'intégrité. Il est également possible de définir ses propres contraintes, par exemple, en limitant les valeurs d'un attribut particulier aux valeurs contenues dans un ensemble déterminé.

Dans le schéma d'exemple, les identifiants sont les groupes qui se trouvent dans la troisième partie des types d'entités et qui sont précédés de **id :**. Le terme "groupe" désigne un ensemble non vide d'attributs (exemple : **ncli** dans l'identifiant de **Client** et **ncom** dans l'identifiant de **Commande**) et de relations (exemple : **en.Produit** et **de.Commande** qui forment l'identifiant de **Detail**). Un autre exemple de contrainte d'intégrité est celle constituée par la relation **passe** qui fait en sorte qu'une commande a toujours un et un seul client. Cette construction particulière donne lieu à une contrainte référentielle, dans le

modèle relationnel. On dit alors qu'une commande référence un client.

Nous terminons ici le rappel sur le modèle ERA. Une présentation plus détaillée est disponible dans [26].

Schémas logiques et physiques

Nous allons brièvement rappeler les principaux concepts des modèles logique et physique. L'ouvrage de référence pour ceux-ci est [29].

"Le schéma logique d'une base de données est la traduction du schéma conceptuel en structure de données propres à une famille de technologie."

La famille de technologies que nous considérerons ici est celle des bases de données relationnelles. Le modèle logique que nous utiliserons est le modèle relationnel. Afin d'illustrer les différents concepts, nous utiliserons l'exemple à la figure 3.2, qui est le modèle logique relationnel correspondant au schéma conceptuel de la figure 3.1.

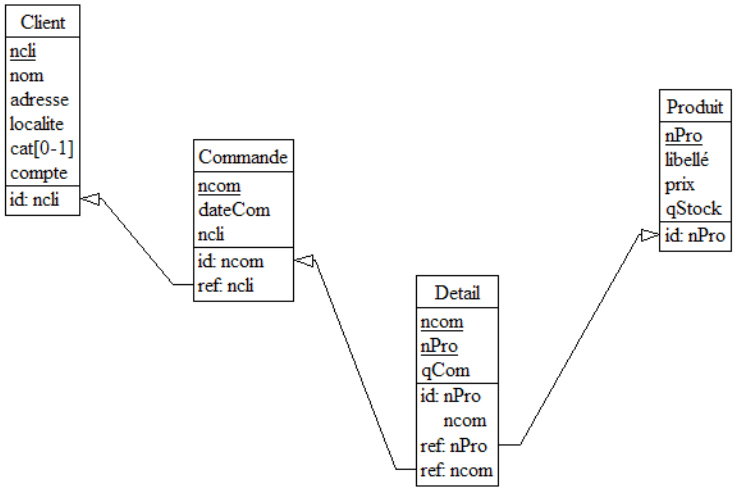

FIG. 3.2 – Exemple de schéma logique

Les éléments principaux du modèle logique sont les types d'enregistrements. Ceux-ci représentent la structure de tous les enregistrements du même type dans la base de données. Un enregistrement représente une entité conceptuelle, une partie d'une entité ou encore plusieurs entités. Un type d'enregistrements représente donc un, une partie ou plusieurs types d'entités conceptuelles. Dans l'exemple, les types d'enregistrements sont : **Client**, **Commande**, **Produit** et **Detail**. Les types d'enregistrements contiennent des champs, qui seront instanciés pour chaque enregistrement de ce type. Dans la représentation graphique du modèle relationnel, les champs sont décrits dans la deuxième partie des types d'enregistrements. Lorsque le modèle conceptuel est assez simple, les champs du modèle logique correspondent aux attributs du modèle conceptuel. Par exemple, les champs du type d'enregistrements **Client** sont : **ncli**, **nom**, **adresse**, etc. Un champ, tout comme un attribut, peut être monovalué ou multivalué, atomique ou composé, facultatif ou obligatoire et stable ou modifiable. Chaque champ possède également un domaine, c'est-à-dire un type de valeur technique (exemple : entier, décimal, caractère, etc.) ou défini par l'utilisateur (exemple : numéro de téléphone, adresse, etc.). Il existe d'autres propriétés des champs et en particuliers celles décrivant les champs multivalués qui, dans la base de données, représentent une collection de valeurs. Pour plus de détails sur ces autres propriétés, nous invitons le lecteur

à consulter l'ouvrage de référence.

Le dernier élément du modèle relationnel que nous allons voir ici est le groupe. Un groupe appartient à un type d'enregistrements et se situe, dans la représentation graphique, dans la troisième partie du type d'enregistrements auquel il appartient. Un groupe exprime une contrainte sur les champs qui le constituent. Il peut s'agir d'une contrainte d'unicité, un identifiant primaire ou secondaire, qui équivaut à celle du modèle ERA. Il peut également s'agir des contraintes d'existence de type **coex** (coexistence), **excl** (exclusion mutuelle), etc., également présentes dans le modèle ERA. La contrainte référentielle est particulière au modèle relationnel et est désignée par **ref** dans le modèle. Elle impose que toutes les valeurs des champs du groupe existent déjà dans l'identifiant primaire ou secondaire du type d'enregistrements référencé. Dans l'exemple, **Commande** référence **Client**. Cela implique que chaque valeur de **ncli** de **Commande** se trouve dans **ncli** de **Client**. La contrainte référentielle est une des contraintes les plus utilisées dans le modèle relationnel, avec les identifiants.

Ceci termine notre introduction au modèle relationnel.

Le modèle suivant est le modèle physique qui se défini comme suit :

"Le modèle physique est l'interprétation particulière du schéma logique dans une technologie de cette famille. Il est généralement constitué des objets du schéma logique augmentés de constructions techniques d'implémentation."

Les paramètres particuliers du schéma physique décrivent, par exemple, les techniques de représentations des valeurs, les caractéristiques des espaces de stockage, la gestion de la concurrence, etc. Par la suite, nous n'exploiterons pas le modèle physique et resterons au niveau d'abstraction du modèle relationnel et du modèle ERA étendu.

Transformation d'un schéma conceptuel en un schéma physique

Pour cette introduction aux tranformations de schémas, nous utiliserons [28] comme ouvrage de référence.

Nous avons introduit le modèle conceptuel ERA et le modèle logique relationnel. Il reste la question de la transformation d'un modèle ERA en un modèle relationnel. Cette transformation se fait au moyen d'opérateurs de transformations de schéma. Avec ces opérateurs, il est possible de transformer un schéma conceptuel dans une forme telle qu'elle peut être traduite en un schéma relationnel. Ces opérateurs sont nombreux et permettent d'obtenir des schémas équivalents, exprimés dans différentes formes. Ces opérateurs agissent aussi bien sur les types d'entités et les types d'associations que sur les attributs et les contraintes d'intégrité. Ils existent quatre opérateurs de base réversibles. Ceux-ci permettent de :

- transformer un type d'associations en un type d'entités ;
- transformer un type d'associations en un attribut ;
- muter un attribut en un type d'entités représentant les valeurs de l'attribut ;
- changer un attribut en un type d'entités représentant les instances de l'attribut.

Nous ne verrons ici que les transformations principales qui permettent de passer d'un schéma ERA à un schéma relationnel.

Un type d'entités simple ne pose en général pas de difficultés lorsqu'on veut le traduire dans un modèle relationnel. Le type d'entités devient, en effet, un type d'enregistrements portant le même nom. Les attributs du type d'entités deviennent les champs du type d'enregistrements. L'attribut **A**, identifiant le type d'entités, devient le champ **A** du type d'enregistrement et constituera l'identifiant du type d'enregistrements. La figure 3.3 présente une telle transformation.

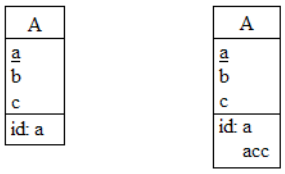

FIG. 3.3 – Transfomation d'un type d'entités (objet de gauche) simple en un type d'enregistrements (objet de droite).

Les relations plusieurs-à-plusieurs ne sont pas représentables dans le modèle relationnel. Pour les représenter, il faut d'abord transformer le type d'associations en un type d'entités avec le premier des quatre opérateurs principaux. Si le type d'association est N-aire, il y aura N nouvelles relations un-à-plusieurs entre le nouveau type d'entités et les N types d'entités d'origine. L'identifiant du nouveau type d'entités sera son ancien identifiant augmenté des N nouvelles relations. La figure 3.4 illustre cet opérateur. Un fois les transformations effectuées, le schéma ERA obtenu peut être traduit en un schéma relationnel. Les relations un-à-plusieurs du schéma conceptuel sont traduites en des contraintes référentielles dans le schéma relationnel, comme présenté à la figure 3.4. Pour ce faire, on utilise le deuxième grand opérateur qui transforme un type d'associations en un attribut.

Les techniques de transformations que nous venons de voir donnent un exemple du lien existant entre les schémas conceptuels ERA et les schémas logiques relationnels.

Qualité du schéma conceptuel

Nous n'étudierons pas, dans ce mémoire, la qualité du schéma. Les règles définissant ce qu'est un schéma de qualité font partie d'un aspect non repris dans ce mémoire. Néanmoins, pour étudier le rapport entre le schéma des données et le reste de l'application, il est nécessaire de caractériser le schéma. Nous verrons donc certains attributs du schéma. Ces attributs seront des attributs internes, afin de définir des modèles les plus objectifs possibles. Ces attributs sont, par exemple, le nombre de types d'entités, d'attributs, de relations binaires ou N-aires, etc. Ces éléments constituent un ensemble d'éléments formalisés dans un schéma. Afin de ne pas s'étendre dans des constructions trop complexes, nous n'étudierons pas non plus les contraintes représentant des règles business, définies par un "utilisateur" et fournies généralement sous forme de règles mathématiques dans une annotation.

Différentes mesures et modèles vont être développés par la suite afin d'obtenir des indicateurs sur le schéma. Il faudra, par exemple, pouvoir quantifier la taille d'un schéma, ou encore définir son niveau de complexité.

3.1.2 Implémentation des contraintes

Lors de la présentation des modèles ERA étendus et des modèles relationnels, à la partie 3.1.1, il a été question des contraintes d'intégrités. Ces contraintes sont assez proches du conceptuel au relationnel, mais que deviennent-elles lors de l'implémentation ? S'il s'agit d'une base de données relationnelle, elles peuvent être soit intégrées automatiquement par le SGBD [1], soit être mises en place par un mécanisme SQL écrit par un programmeur ou encore être gérées par du code autre que SQL se trouvant hors du SGBD. Les identifiants et les contraintes référentielles sont généralement vérifiées par le SGBD. Il suffit alors de

[1] SGBD : Système de gestion de base de données.

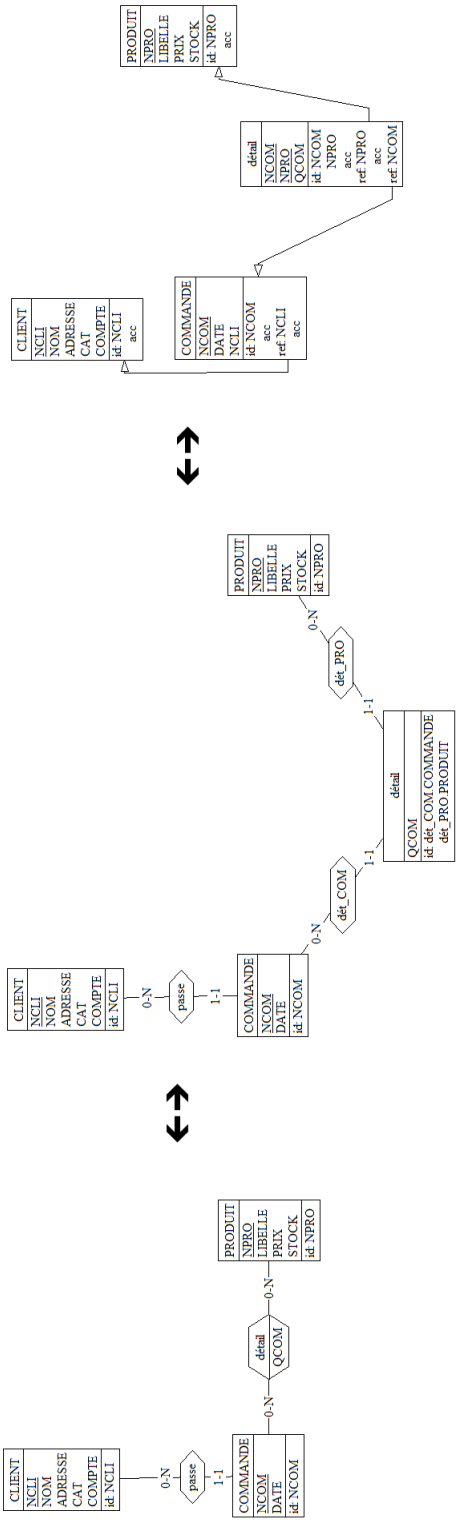

FIG. 3.4 – Transfomation d'un schéma ERA en un schéma relationnel avec le premier grand opérateur.

signaler, avec quelques mot-clés SQL, le type de contraintes ainsi que les colonnes et les tables concernées. Les autres contraintes d'intégrité, telles que les contraintes d'existence, celles sur les domaines de valeurs ou encore celles définies par l'utilisateur peuvent être mises en place soit par un mécanisme SQL, si le SGBD supporte un tel mécanisme, soit dans le code externe à la base de données, comme par exemple du code JAVA. Pour le premier cas, on utilisera un check, qui vérifie les conditions simples, un trigger, qui peut vérifier les conditions allant des plus simples aux plus compliquées et entraîner des modifications dans la base de données, etc.

Ces explications, assez brèves, montrent bien le problème de l'analyse du code SQL d'une application. D'une part, il faut considérer les requêtes mais également les mécanismes déclenchés par ces requêtes. D'autre part, le code SQL est principalement déclaratif et ne peut donc être analysé par les méthodes s'appliquant aux langages procéduraux. Les mécanismes SQL, comme les triggers, permettent toutefois d'écrire en même temps dans le paradigme procédural et dans le déclaratif. Nous ajouterons que dans un système d'accès aux fichiers COBOL, par exemple, les contraintes d'intégrité ne peuvent être implémentées qu'au travers du code COBOL de l'application. Il en est de même pour les SGBD ne supportant pas certains mécanismes SQL. Ces mécanismes non supportés peuvent aller de certains types de triggers, aux clés étrangères, comme c'est le cas pour certaines versions du SGBD *MySql*.

Nous ne présenterons pas ici le langage SQL et les différents mécanismes SQL que nous venons d'aborder. Une introduction au langage SQL est diponible dans [25]. La définition du langage *SQL92*, que nous avons utilisé comme référence du langage SQL dans ce mémoire, est disponible dans [22]. Pour une présentation complète du langage SQL ainsi qu'une description des différents mécanismes, nous redirigeons le lecteur vers [24].

3.2 Développement avancé du problème

Comme nous en avons parlé précédemment, peu d'auteurs ont abordé le problème auquel nous nous intéressons dans ce mémoire. Les raisons de cette lacune ne sont pas évidentes, même si la plus intuitive pourrait être que le code SQL nécessaire à la mise en place du schéma dans le SGBD et le code des requêtes SQL ne représentent bien souvent qu'une infime partie du logiciel en terme de SLOC. A l'exception, peut-être, des bases de données actives dans lesquelles les contraintes business sont représentées sous formes de triggers et d'autres mécanismes, une fois le schéma mis en place, la base de données passe en arrière plan. Pourtant, il existe un lien fort entre les schémas conceptuels, logiques et physiques. On obtient en effet, le schéma physique du logique qui est lui même dérivé du conceptuel. De même, il est possible, dans une certaine limite, d'obtenir le schéma conceptuel à partir du schéma physique. Le schéma conceptuel est également utilisé pour créer la structure des données dans toute l'application et donc à d'autres niveaux que dans la base de données (exemple : les objets business en JAVA). Comme nous l'avons cité, il existe un lien entre la taille du schéma conceptuel et celle de l'application. Nous pouvons donc pousser plus loin et étudier la corrélation entre certaines caractéristiques du schéma de la base de données et celles des accès à la base de données, ainsi qu'entre ces deux dernières et les caractéristiques de l'application. Ce raisonnement est représenté à la figure 3.5. Les requêtes SQL se situent généralement entre la base de données et, par exemple, JAVA. Celles-ci vont également constituer une source d'informations importantes puisqu'elles seront le moyen de communication entre la base de données et une très grande partie de l'application. Nous allons illustrer ces problèmes et intuitions avec les exemples suivants.

On se souvient encore des systèmes d'accès aux fichiers très utilisés il y a maintenant plus d'une quinzaine d'années et pour lesquels les contraintes étaient implémentées sous

FIG. 3.5 – Représentation des influences.

forme de code procédural (exemple : fichiers COBOL). Les bases de données relationnelles, sur lesquelles nous travaillerons ici, ont permis de mettre en place certaines contraintes, comme les clés étrangères, de manière simple au moyen de mécanismes intégrés. D'autres contraintes, comme les domaines de valeurs, ont pu être implémentées au travers de triggers ou de checks. Ces changements ont bien évidemment amélioré la qualité générale des applications mais ces différences ne semblent pas avoir été étudiées jusqu'à maintenant. De nombreux systèmes anciens, utilisant les fichiers COBOL, sont encore utilisés de nos jours. Ces systèmes forment parfois le composant informatique principal des sociétés. Il ne s'agit donc pas d'un exemple dépassé, mais bien d'un cas concret de l'informatique. Ces anciens systèmes, appelés aussi "legacy systems", sont devenus trop lourds, trop imposants et surtout trop importants pour les sociétés qui les utilisent pour qu'ils soient entièrement reconstruits. Détecter ses failles en terme de qualité peut donc se révéler utile afin d'estimer certains coûts. D'autre part, il existe des applications récentes qui, pour des raisons de financement, utilisent des SGBD ne permettant pas de mettre en place ces contraintes. Celles-ci sont donc implémentées sous forme de code procédural. Il ne s'agit que d'une partie du problème. La question de la qualité est également valable pour les applications qui utilisent justement les bases de données relationnelles.

Bien entendu, les contraintes d'intégrité d'une base de données ne sont pas les seuls éléments qu'il nous semble possible d'exploiter. Les requêtes aux bases de données, qui transmettent une information sur d'une part, les données entrantes et sortantes de la requête et d'autre part, sur la complexité de cette dernière, peuvent amener de nouvelles informations sur l'application. Nous aborderons également certaines mesures associées actuellement aux bases de données et qui pourraient être adaptées dans le but de fournir un résultat plus critique, plus précis. Par exemple, le fait de compter un read, un write, etc., comme valant 1, était valable pour des lectures dans des fichiers, mais qu'en est-il dans une base de données relationnelle ? Cette taille de 1 correspond-elle à quelque chose de concret dans l'application globale ?

Nous avons également présenté la cohésion dans le chapitre précédent qui peut se rapporter aux données. Or, ces données peuvent être stockées dans la base de données et doivent donc être extraites par des requêtes. Nous tenterons donc ici d'exploiter au mieux les informations implicites se trouvant dans ces éléments et de proposer un ensemble de techniques pouvant soit être intégrées à des méthodes existant, soit en former de nouvelles. Les sujets abordés dans ce mémoire seront principalement les attributs internes du logiciel liés à la maintenabilité. Lors du développement d'une nouvelle mesure ou d'un nouveau modèle, il est important que ces derniers confirment l'intuition obtenue en étudiant l'objet considéré. De même il est primordial de ne pas chercher à combiner trop de paramètres différents afin d'éviter de créer un modèle complexe et peu significatif. Cette critique a été faite à de nombreux modèles de mesures en génie logiciel. Les modèles qui vont être développés par la suite resteront donc les plus proches possibles de l'intuition, ou encore des connaissances implicites liées aux bases de données et au code en général. Le premier problème a résoudre est celui de la caractérisation des schémas de données. Un fois ces schémas caractérisés, nous pourrons procéder à l'étude des accès à la base de données. Là encore, nous devrons chercher à caractériser ces accès. Nous le ferons en terme de complexité. Finalement, les caractéristiques de l'application, dans son ensemble, vont être

analysées et nous tenterons de voir s'il existe un rapport entre les attributs des schémas et des accès aux données vues précédemment et ceux du logiciel.

3.3 Caractérisation du schéma des données

3.3.1 Taille du schéma

Intuitivement, la taille du schéma est vue comme le nombre d'éléments que celui-ci contient. Une méthode de COCOMOII [7] calcule la taille en fonction du nombre de bytes ou de caractères constituant la base de données. Il s'agit d'une façon naturelle de caractériser la taille mais le résultat dépend de la façon dont on utilise la base de données. La première question à laquelle il nous faut répondre est : quels sont les éléments d'un schéma ? On trouve dans le schéma conceptuel des types d'entités, d'associations, etc. Ces éléments ont été décrits dans l'introduction au schéma conceptuel, à la section 3.1.1. Ces éléments sont donc nombreux mais sont internes au schéma. De ce fait, ils peuvent être comptés objectivement.

Un premier modèle serait de prendre la taille du schéma comme étant égale au nombre de types d'entités. Le résultat est certainement trop approximatif car le nombre d'attributs appartenant à un type d'entités varie. Cela aurait pour conséquence que deux schémas distincts composés d'un seul type d'entités chacun, mais dont le nombre d'attributs dans le premier schéma est de 5 et dont le deuxième schéma comprend 30 attributs obtiennent la même taille.

Le même problème se pose si on ne prend en compte que le nombre de type d'associations. Toujours dans un cas purement académique, si on considère un premier schéma contenant deux types d'entité liés par une seule relation et un autre schéma contenant six types d'entités n'ayant aucune relation entre elles, le premier schéma serait alors plus grand que le deuxième. Ce modèle ne tiendrait pas compte de tous les paramètres.

Pour avoir un "bon" modèle de la taille, il nous faut prendre en compte l'ensemble des caractéristiques du schéma. Pour ce faire, nous allons d'abord identifier les composants des schémas sur base du méta-modèle à la figure 3.6. Ce méta-modèle schématise la structure des données contenue dans un schéma entités-associations de DB-MAIN. Dans ce modèle, nous ne tiendrons pas compte des types d'entités **DBMProcessingUnit**, **ProcessingUnitOwner** et **DBMCollection**. Les éléments restant expriment la structure de la plupart des schémas entités-associations. Il nous reste à choisir les éléments qui font la taille d'un schéma conceptuel. Dans la section suivante, nous présenterons le problème de la complexité du schéma, et nous souhaitons ici rendre la taille et la complexité du schéma les plus indépendantes possibles. L'idéal est donc d'utiliser, pour caractériser la taille du schéma, des types d'entités du méta-modèle différents de ceux utilisés pour caractériser la complexité du schéma. Bien entendu, ces types d'entités ne peuvent être entièrement indépendants, entre autres à cause des relations entre certains types d'entités du méta-modèle. Prenons pour exemple les rôles, représentés par le type d'entités **DBMRole** et les types d'associations, représentés par le type d'entités **DBMRelationshipType**. Les types d'associations vont servir à évaluer, en partie, la taille du schéma. Les rôles vont être utilisés pour évaluer un aspect de la complexité du schéma. On observe que ces deux types d'entités ne sont pas entièrement indépendants dans le méta-modèle. Les rôles ne peuvent exister sans un type d'associations. Le méta-modèle précise qu'il faut au moins un type d'associations pour tous les rôles du schéma. En pratique, un type d'associations est généralement binaire, parfois ternaire et possède rarement une cardinalité supérieure. Un cas pratique aura donc, généralement, un nombre de rôles par type d'associations situé entre 2 et 3. Une autre remarque est que les types d'associations ont, en pratique, au moins deux rôles, même si le méta-modèle accepte les types d'associations sans rôle et unaire. Le

méta-modèle ne montre qu'une faible relation entre la taille et la complexité. L'étude par la pratique montre quant à elle, une relation assez forte entre certains éléments. La taille et la complexité ne sont donc pas indépendants. Cette relation entre la taille et la complexité est néanmoins naturelle. En effet, il est normal que le nombre d'éléments complexes augmente en même temps que la taille du schéma. La complexité " moyenne " sera quant à elle, assez constante.

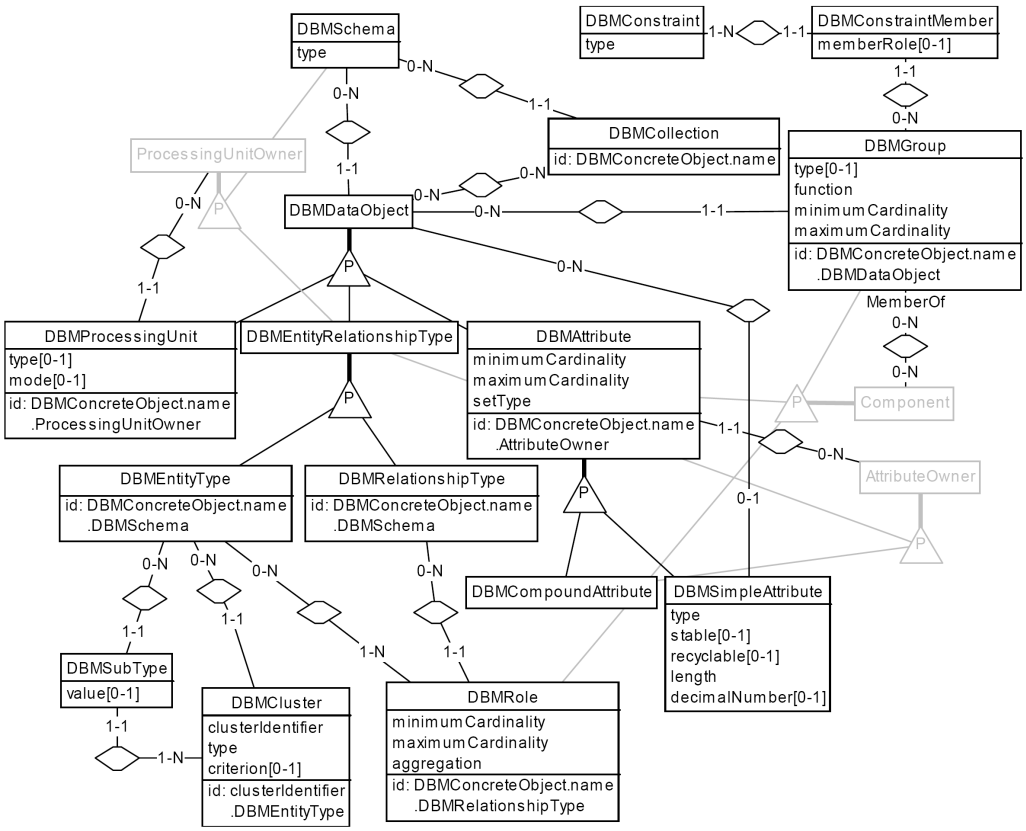

FIG. 3.6 – Vue données du repository DB-MAIN pour JIDBM.

Afin de calculer la taille, nous utiliserons les éléments appartenant à la classe des **DBMDataObject**. Étant donné les caractéristiques des deux relations d'héritage partant de **DBMEntityRelationshipType** et **DBMAttribute**, nous utiliserons les types d'entités suivants :

- **DBMEntityType** : Représente les types d'entités des schémas conceptuels ;
- **DBMRelationshipType** : Représente les types d'associations des schémas conceptuels ;
- **DBMCompoundAttribute** : Représente les attributs composés qui peuvent posséder des groupes ou d'autres attributs ;
- **DBMSimpleAttribute** : Représentent les attributs simples.

Ce choix est arbitraire et peut être contesté mais il permet d'évaluer la taille d'un schéma selon des règles très précises. Le problème restant est celui de l'échelle de valeur servant au comptage. Nous avons ici choisi de compter chacun de ces éléments comme valant une unité. Ce choix est contre-intuitif, mais il est justifié par le fait que dans le méta-modèle à la figure 3.6, tous les éléments que nous venons de citer sont des **DBMDataObject**. Par les relations d'héritage présentes dans le schéma, le type d'entités **DBMDa-**

taObject regroupe tous les éléments principaux du schéma. De plus, étant donné ces choix, la mesure de la taille d'un schéma présente plusieurs avantages. Le premier est que l'unité de mesure rend comparable la taille de différents schémas. Le résultat de la mesure est aussi unique et simple. Par "simple", nous exprimons le fait que le résultat n'est pas composé de plusieurs valeurs mais d'une seule. Nous aurions eu plusieurs valeurs si nous avions, par exemple, distingué les unités des types d'entités, de celles des types d'associations dans les schémas. Le résultat est unique pour chaque schéma, car la mesure est objective. Elle est également automatisable, cela au moyen d'un encodage formel du schéma, suivi d'un comptage des éléments désirés.

Pour illustrer cette mesure de la taille, nous allons utiliser le schéma B.1, de l'annexe B. Cet exemple provient des ceux fournis par l'outil DB-MAIN. Nous avons compté dans le schéma, 12 types d'entités, 12 types d'associations, 38 attributs simples et 5 attributs composés. La taille totale du schéma est donc de 67, c'est-à-dire, la somme des éléments que nous venons de citer.

3.3.2 Complexité du schéma

Cette partie présente une mesure permettant d'évaluer la complexité des schémas ERA. Nous commencerons par voir ce qui correspond intuitivement au concept de complexité afin de définir une mesure de celle-ci pour les schémas ERA. Les complexités qui nous intéressent ici sont la complexité structurelle et la complexité cognitive du schéma. Lorsqu'on évalue la structure du code, comme nous l'avons présenté au point 2.2.2, ce dernier, s'il s'agit de code procédural, peut être représenté sous la forme d'un graphe orienté. Il n'est malheureusement pas possible de faire de même avec le schéma conceptuel des données. La raison est que, bien qu'un schéma soit un graphe celui-ci ne comporte pas de noeuds de décision contrairement aux représentations du code procédural sous forme de graphe. Or, c'est sur ces noeuds de décision qui représentent par exemple des **if-then-else**, des **while**, etc., que s'appuient les mesures de complexité du code procédural. La complexité peut, toutefois, être définie en analysant la structure du schéma. Plusieurs études présentées dans le chapitre consacré aux modèles existant ont montré qu'il est possible de quantifier la complexité d'une structure de données comme un diagramme de classe UML [20]. Pour ce qui est de ce que nous avons appelé la complexité cognitive, nous nous appuyerons sur la complexité structurelle du schéma pour la définir. Elle permettra de classer certains éléments complexes des schémas dans un ordre de grandeur. Nous abordons la complexité des schémas afin de fournir un outil qui aidera à déterminer quels sont les liens entre la complexité du schéma et celle du code SQL.

On trouve généralement qu'un schéma est plus complexe lorsque celui-ci est plus difficile à comprendre, mais il s'agit alors d'une complexité cognitive et non plus structurelle. Ces deux types sont toutefois fortement liés. La complexité structurelle devrait normalement être un attribut interne au schéma et devrait pouvoir être quantifiée objectivement, contrairement à la complexité cognitive qui fait intervenir des conditions extérieures comme l'expérience et la formation de la personne analysant le schéma, le design du schéma, etc. Nous ne séparerons plus entièrement ces deux types de complexité. Nous identifierons d'abord les structures complexes d'un schéma entités-associations. Nous établirons ensuite des classes de complexité à partir des éléments identifiés.

La première étape va donc consister en l'identification des éléments qui peuvent influencer la complexité d'un schéma. Nous avons, dans la partie 3.3.1, choisi de rendre la complexité et la taille du schéma les plus indépendantes possibles. Toujours sur base des éléments du méta-modèle à la figure 3.6, nous allons définir les éléments complexes d'un schéma. Intuitivement, le nombre de contraintes de types at-least-1 ou de coexistence, le nombre de rôles pour un type d'associations, le type des relations (One-to-Many, One-to-One, Many-to-Many) sont des exemples d'éléments constituant la complexité d'un schéma.

Dans le méta-modèle, ces éléments sont les types d'entités en relation avec le type d'entités **DBMDataObject** ou ses descendants. Nous définirons donc la complexité du schéma sur base des groupes, des rôles, des relations IS-A et de la composition des attributs.

Les éléments influençant la complexité des schémas sont les suivants [2] :

- **DBMCluster** : indique une spécialisation, c'est-à-dire un IS-A, pour un type d'entités. Cet objet est lié au super-type de la relation ainsi qu'aux types d'entités indiquant les sous-types ;
- **DBMSubType** : indique les sous-types liés dans un IS-A. Cet objet est lié au **DBMCluster** ainsi qu'aux types d'entités constituant les sous-types dans la relation IS-A ;
- **DBMRole** : indique un rôle dans un type d'associations ;
- **DBMGroup** : un groupe est utilisé pour représenter une contrainte telle qu'un identifiant, une contrainte d'existence, etc. Il peut être composé d'attributs, de rôles et d'autres groupes. Il est rattaché à un type d'entités, un type d'associations ou encore un attribut composé. L'attribut **function** représente le type de contrainte ;
- **DBMConstraintMember** : effectue la liaison entre plusieurs groupes lorsque la contrainte est une contrainte inter-groupe ;
- **DBMConstraint** : spécifie le comportement ensembliste d'une contrainte inter-groupe, comme par exemple, l'inclusion, l'égalité, etc. ;
- **AttributeOwner** : indique le propriétaire de l'attribut. Ce propriétaire peut être un type d'entités, un type d'associations ou encore un attribut composé.

Nous tiendrons également compte du nombre de ces éléments ainsi que de certaines des relations entre ces éléments et les autres types d'entités du schéma. Notre mesure de la complexité peut être introduite de la façon suivante. Nous pouvons imaginer une vue en hauteur du schéma. De cette vue, nous distinguons des structures, sans connaître leur sens réel. Nous pouvons par exemple, voir un type d'associations avec 3 rôles. Il va nous sembler plus complexe qu'un autre type d'associations avec 2 rôles. De même, en ce qui concerne les groupes, plus ils contiendront d'éléments, plus il nous paraîtront complexes. Dans les relations IS-A, nous distinguerons un IS-A, sans savoir de quel type il est, mais nous verrons combien de sous-types directs se trouvent dans la relation[3]. Finalement, nous pourrons observer le niveau de profondeur des attributs composés. Cette interprétation de la complexité structurelle des schémas ne tient donc pas compte de la sémantique des structures choisies. Cette sémantique est représentée par les attributs des types d'entités **DBMCluster**, **DBMSubType**, etc du méta-modèle.

Il n'est bien entendu pas question ici de comparer la complexité engendrée par le nombre de rôles et celle engendrée, par exemple, par les IS-A. Nous devrions alors mettre des structures de natures très différentes sur un même pied d'égalité. Il est par contre possible de définir des "familles" d'éléments complexes. Les éléments d'une même famille peuvent être comparés entre-eux. L'exemple le plus simple est celui des rôles. Nous utiliserons la figure 3.7 comme exemple. Il est clair que les rôles du type d'associations **emp-clôturé** forment une structure plus complexe que la structure du type d'associations **répondant**. Voici un autre exemple, toujours basé sur la figure 3.7, qui touche aux contraintes du schéma. Les contraintes sont, dans le méta-modèle, représentées par des groupes. Les identifiants sont donc des groupes. Nous avons considéré ici la complexité

[2] Une définition complète de ces éléments est disponible dans [37].

[3] Par sous-type direct, nous faisons ici référence uniquement aux types d'entités se trouvant au premier niveau de profondeur dans la relation.

structurelle de ces groupes comme étant le nombre d'éléments (groupes, attributs ou rôles)[4] présents dans le groupe. Ces éléments sont représentés dans le méta-modèle par le type d'entités **Component**. En ce qui concerne les relations IS-A, la complexité de leur structure dépend du nombre de sous-types associés pour la relation IS-A. Ces sous-types correspondent au type d'entités **DBMSubType** dans le méta-modèle.

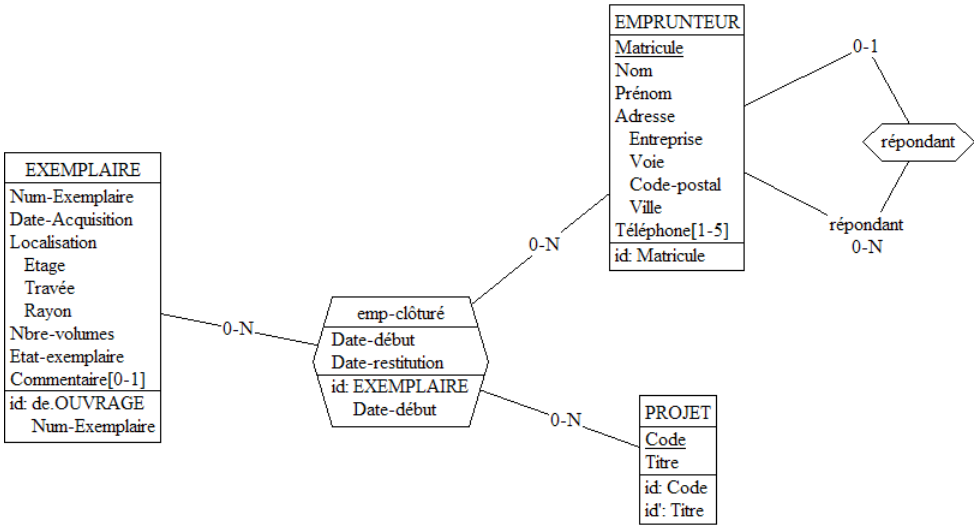

FIG. 3.7 – Illustration de différentes structures possibles dans un schéma conceptuel.

Pour ces 3 familles de structures complexes, les groupes, l'arité des types d'associations et les IS-A et le nombre de sous-types associés, nous pouvons distinguer une complexité dite cognitive. Cette complexité va dépendre de la structure et de la sémantique. La notion de complexité va donc devenir la difficulté avec laquelle un schéma peut être compris. Nous ne pourrons développer ici un ordre général complet sur tous les éléments complexes. Comme nous l'illustrerons au travers d'un exemple, certains éléments sont trop proches et ne peuvent donc être classés dans un ordre précis. Nous allons tout d'abord étudier les groupes. Pour ceux-ci, nous utiliserons la fonction du groupe afin de les classer. La fonction est représentée dans le méta-modèle par l'attribut **function** du type d'entités **DBMGroup**. La fonction du groupe peut avoir une des valeurs suivantes : pas de contrainte, identifiant primaire, identifiant secondaire, at-least-one, coexistence, exclusion ou contrainte définie par l'utilisateur. [5] Parmi ces éléments, le plus courant est la contrainte d'identifiant. Celle-ci est présente dans la plupart des types d'entités d'un schéma. Les éléments de type coexistence, exclusion et at-least-one impliquent une contrainte sur la valeur des éléments du schéma, et forment les contraintes d'existence. Ces contraintes

[4]Pour garder une certaine simplicité, nous avons choisi de ne pas aborder les problèmes concernant les éléments qui composent les groupes. En effet, un groupe peut composer un autre groupe. Nous pouvons alors être dans une situation où, pour deux attributs **a** et **b**, nous pourrions avoir le groupe **1**, contenant **a** et **b**. Nous pourrions également avoir le groupe **2**, composé de **1** ainsi qu'un groupe **3**, contenant **1**, **a** et **b**. En résumé, nous aurions **1** = {**a**,**b**}, **2** = {{a,b}} et **3** = {a,b,{a,b}}. Il s'agit donc d'un problème de comparaison entre les composants d'un groupe et on peut se demander si **1** et **2** ou encore si **2** et **3** sont équivalents. Par simplicité, nous considérerons, dans un même groupe, les attributs, les rôles et les groupes, quels que soient les éléments qu'ils contiennent, au même niveau.

[5]Nous n'avons pas tenu compte des clés d'accès qui n'apparaissent normalement que dans les schémas physiques. Toutefois, si celle-ci devait être classée, nous la situerions entre l'absence de contrainte et la contrainte d'identifiant. En effet, la clé d'accès n'implique pas de contrainte particulière sur la valeur des éléments du schéma et par rapport aux identifiants pour lesquels on écrit toujours une clé d'accès, la contrainte "clé d'accès" est moins complexe.

nous ont semblé plus difficiles à interpréter que celles indiquées par les identifiants. Pour cette raison, elles sont considérées comme étant plus complexes que l'identifiant. Parmi les trois contraintes d'existence, nous ne distinguerons pas d'ordre. Finalement, les contraintes définies par l'utilisateur ne seront pas classées car elle sont trop variables. Voici donc l'ordre des contraintes des groupes :

pas de contrainte < identifiant primaire et secondaire < coexistence, at-least-one et exclusion

Le problème restant est de définir un ordre de grandeur de complexité, au sens général du terme, sur les groupes. Nous allons ici combiner la complexité structurelle et l'ordre des contraintes. L'ordre général va d'abord se faire sur base de la taille du groupe (le critère de complexité structurelle). Pour une même complexité structurelle, les éléments seront ordonnés selon l'ordre des contraintes défini précédemment. Ce classement a l'avantage d'être simple et de se baser sur la structure et sur l'aspect compréhension. Il nous paraît aussi assez intuitif. Il a comme désavantage de ne pas évaluer numériquement la complexité et de ne pas permettre la comparaison d'éléments qui n'ont pas une même structure. Il s'agit là d'un choix que nous justifions par la question suivante : un identifiant constitué de quatre attributs, rôles ou autre groupes, est-il plus ou moins complexe qu'une contrainte d'exclusion à deux éléments ? Il ne s'agit que d'un exemple parmi d'autres. Ces questions ne peuvent être éclaircies que par des études de cas et le résultat dépendra trop, selon nous, de la formation et des capacités de l'individu analysant le schéma.

L'évaluation de la complexité des rôles et des types d'associations est construite selon un procédé similaire. Nous ne comparerons entre eux que les constructions qui ont une complexité structurelle similaire et donc les types d'associations ayant une même arité. Il est ensuite possible, à partir de celle-ci, de définir un ordre de complexité en fonction de la cardinalité des rôles. Nous utiliserons une relation binaire afin d'illustrer cet ordre. Un type d'associations binaire peut être de trois classes fonctionnelles différentes, représentées par la cardinalité maximale des rôles. Dans le méta-modèle, cette information est donnée par l'attribut **maximumCardinality** du type d'entités **DBMRole**. Ces trois classes sont : un-à-un, un-à-plusieurs ou plusieurs-à-plusieurs. Les trois exemples suivants, aux figures 3.8, 3.9 et 3.10, tirés de [25], présentent ces trois classes, pour les relations binaires.

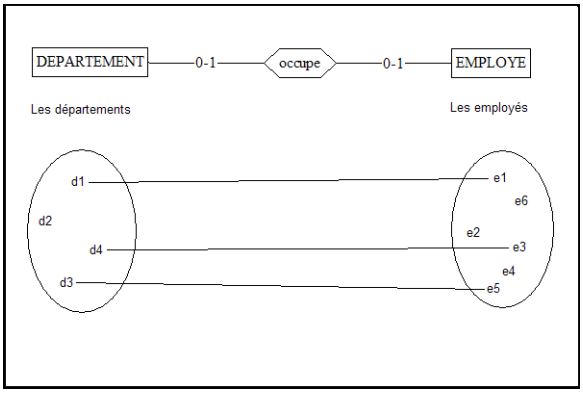

FIG. 3.8 – Relation un-à-un.

Nous pouvons voir, grâce à la représentation graphique des objets du domaine d'application et des relations un-à-un, que cette dernière est plus simple à interpréter que les autres. Elle sera donc l'élément de complexité minimale pour les relations binaires. Ensuite, suivant le nombre de liens existant entre les objets, la relation un-à-plusieurs, est

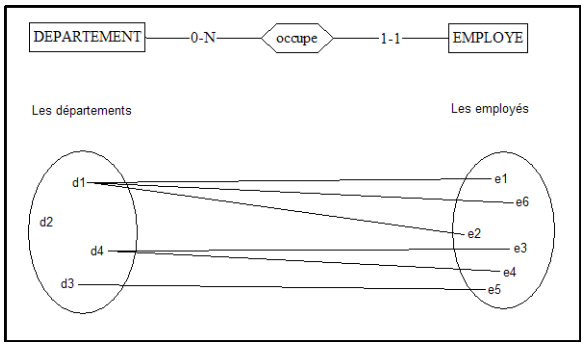

FIG. 3.9 – Relation un-à-plusieurs.

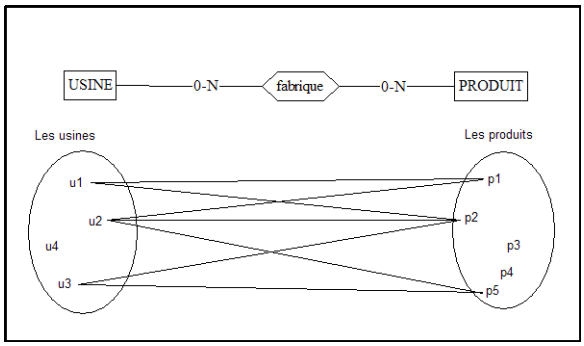

FIG. 3.10 – Relation plusieurs-à-plusieurs.

la deuxième plus complexe. Finalement, la relation binaire la plus complexe est du type plusieurs-à-plusieurs. Les relations d'arité supérieure à 2 utiliseront un classement de complexité similaire, c'est à dire sur base des cardinalités maximales des rôles avec comme élément le moins complexe, la relation ayant tous ses rôles à **1** et comme élément le plus complexe, la relation ayant tous ses rôles à **N**.

La relation **IS-A** est, selon nous, un élément qui augmente la complexité d'un schéma. Bien entendu, elle augmente aussi l'expressivité des schémas ERA. Nous avons choisi de définir la complexité structurelle comme étant le nombre de sous-types déclarés dans la relation. La complexité cognitive va dépendre, quant à elle, du type de la relation, représenté dans le méta-modèle par l'attribut **type** du type d'entités **DBMCluster**. Ce type est une des quatre combinaisons des deux caractéristiques de la relation qui sont "disjoint" et "total". Nous utiliserons la relation ISA à la figure 3.11 comme exemple. Le type de la relation est à indiquer à la place du **?** dans l'exemple. La relation peut donc être partielle et non-disjointe (le **?** est alors remplacé par un blanc), il s'agit de la combinaison qui a la plus faible contrainte mais également de celle qui permet le plus de sous-types possibles au parent **A**. **A** peut être, dans ce cas, de type **B**, **C**, à la fois **B** et **C**, ou encore autre chose. La relation peut être totale et non-disjointe (le **?** est alors remplacé par un **T**). **A** peut alors être de type **B**, **C** ou à la fois **B** et **C**. La troisième combinaison est la relation partielle et disjointe, représentée par un **D**. **A** peut alors être de type **B**, **C** ou un autre type non déclaré dans le schéma. Finalement, la dernière combinaison représente la relation totale et disjointe (**?** est remplacé par **P**), qui exprime la partition. **A** est alors de type **B** ou **C**. Pour une même complexité structurelle, les relations IS-A seront classées en fonction du nombre de sous-types auxquels peut appartenir le parent. Les types d'IS-A seront donc classés de la façon suivante :

total et disjoint (P) < total et non-disjoint (T) et partiel et disjoint (D) < partiel et non-disjoint ( )

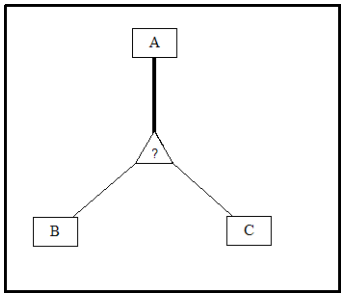

FIG. 3.11 – Relation IS-A

La complexité des attributs composés ne sera définie qu'en terme de structure, contrairement aux groupes, aux relations et aux IS-A. Cette complexité structurelle sera le nombre de niveaux de profondeur existant à partir d'un attribut composé. Le niveau de la racine ne sera pas pris en compte. Ce nombre ne sera considéré que si le parent de l'attribut est un type d'entités ou un type d'associations. Nous sommes conscients que cette description de la complexité peut être encore développée, mais nous nous limiterons à ces éléments. Cette mesure nous a également demandé de faire des choix arbitraires, qui peuvent être contestés. Mais ces choix ont été fait sur base de notre expérience des schémas conceptuels et d'ouvrages présentant ceux-ci. Comme pour la mesure de la taille, nous allons illustrer la mesure de la complexité au travers de la figure B.1.

- Les relations IS-A : L'exemple contient 1 relation IS-A à 1 sous-type et sans contrainte et une relation IS-A à 2 sous-types avec une contrainte de partitionnement ;
- Les types d'associations et les rôles : L'exemple contient 11 relations binaires de type un-à-plusieurs et 1 relation binaire de type plusieurs-à-plusieurs ;
- Les groupes : Le schéma contient 9 groupes à 1 éléments représentant des identifiants et 4 groupes à 2 éléments représentant des identifiants ;
- Les attributs composés : Il y a dans le schéma, 3 attributs composés à 1 niveau et 1 attribut composé à 2 niveaux.

Comme le montre l'exemple, cette mesure de la complexité donne des statistiques simples sur le schéma. Il reste donc à exploiter ces informations, comme nous le ferons dans le chapitre suivant.

3.3.3 Concision et lisibilité du schéma

Nous allons ici brièvement aborder le problème de la concision et de la lisibilité du schéma par rapport à sa complexité. L'un des défauts de la mesure de la complexité des schémas que nous venons de voir est que celle-ci ne donne pas de résultats équivalents pour des schémas équivalents moyennant une transformation faite aux moyens de règles de normalisation. Plutôt que de remettre en question la mesure de complexité, nous avons voulu voir si cette propriété se répercutait sur la concision et la lisibilité du schéma. Nous n'aborderons cette question qu'au travers de deux exemples, aux figures 3.12 et 3.13. Pour ces deux exemples, le schéma de gauche provient de la figure B.1. Le schéma de droite est obtenu au moyen des règles de mutations appliquées par l'outil DB-Main.

La mutation d'un type d'associations en un type d'entités, dans l'exemple 3.12, augmente la taille du schéma ainsi que le nombre d'éléments complexes. La taille passe en effet

de 5 à 7 et le nombre d'éléments complexes passe de 2 (1 relation plusieurs-à-plusieurs et 1 groupe identifiant) à 4 (2 relations un-à-plusieurs et 2 groupes identifiants). Donc, la complexité des relations diminue. On peut dire que la concision diminue puisqu'il faut plus d'éléments pour exprimer la même chose. Il reste donc la lisibilité. Si celle-ci est proportionelle à la concision, elle diminue également. Si par contre, elle est inversément proportionnelle à la complexité des éléments du schéma, elle est modifiée suite à la transformation. Elle augmente car la complexité moyenne des relations diminue mais elle augmente également suite à l'apparition d'un groupe à deux composants. Il ne nous est pas possible ici de quantifier ce changement puisque la complexité des groupes et des relations ne sont pas comparables.

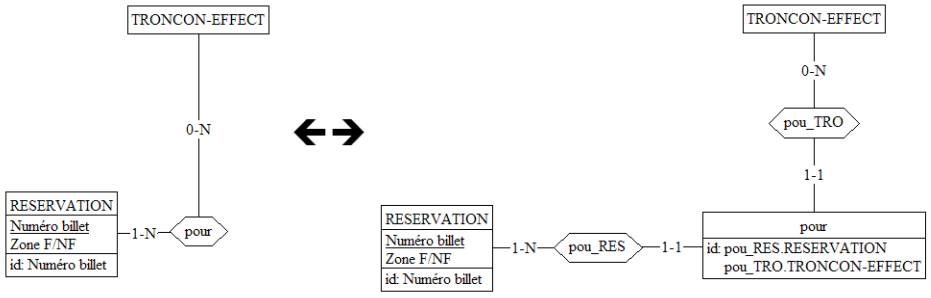

FIG. 3.12 – Exemple de normalisation : Mutation d'un type d'associations en un type d'entités.

L'exemple suivant, à la figure 3.13, concerne la mutation d'un IS-A en types d'associations. La taille passe de 7 à 9. Le schéma d'origine contenait les éléments complexes suivants : 1 groupe identifiant contenant 1 attribut et 1 IS-A de type " partition " avec 2 sous-types. Le schéma obtenu par transformation contient : 1 groupe identifiant contenant 1 élément, 1 groupe de type " exactly-1 " de 2 éléments et 2 relations binaires de type un-à-un. Dans cet exemple, la complexité du IS-A disparaît au profit de celle des groupes et des relations. Il y a une diminution de la concision, puisque la taille augmente. En ce qui concerne la lisibilité, si elle dépend de la complexité des éléments du schéma, nous ne pouvons rien affirmer ici. En effet, le schéma de gauche contient un ISA, qui n'est plus dans le schéma de droite. De même, le schéma de droite contient deux relations exprimées par deux types d'associations qui ne sont pas dans le schéma de gauche.

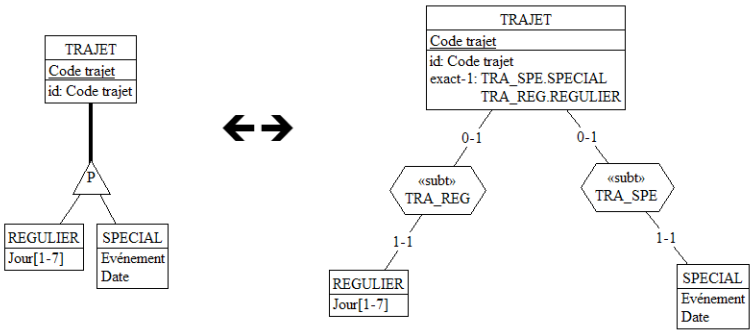

FIG. 3.13 – Exemple de normalisation : Transformation par désagrégation

Il y a donc un aspect des schémas à étudier, suite à ces développements, en ce qui concerne la concision et la lisibilité. Cette étude demande, pour pouvoir être réalisée, une

quantification de la concision et de la lisibilité,que nous ne ferons pas dans le cadre de ce mémoire.

3.3.4 Adaptation au modèle relationnel

La transformation d'un schéma conceptuel en un schéma relationnel entraîne des changements importants. Ceux-ci sont, par exemple, la disparition des types d'associations et des rôles, la disparition des relations ISA, l'apparition de la contrainte référentielle dans les groupes, etc. Nous avons défini les mesures de taille et de complexité sur les schémas ERA. Celles-ci peuvent être également transformées et adaptées au modèle relationnel.

En ce qui concerne la mesure de la taille, nous allons compter les types d'enregistrements de la même façon que les types d'entités. Nous ferons de même avec les champs et les attributs. La transformation du relationnel au conceptuel fait donc disparaître les types d'associations. Ceux-ci n'ont donc pas d'équivalents dans le modèle relationnel. Étant données les règles de normalisation et de transformation [6], les types d'associations vont, en général, être remplacés par des types d'entités et des contraintes référentielles, dans le cas des relations plusieurs-à-plusieurs. Les relations un-à-plusieurs et un-à-un vont être remplacées par une contrainte référentielle. Dans certains cas, les contraintes référentielles peuvent être remplacées par des contraintes d'équivalence. Par exemple, des types d'associations binaires ayant comme cardinalités des rôles [1-1]-[1-N], [0-1]-[1-1], [0-N]-[1-N], etc., donneront lieu à ce type de contraintes.

Il faut également adapter la mesure de complexité au modèle relationnel. La complexité des relations ne pourra plus être comptée, puisque les types d'associations auront disparu. Il en est de même pour les relations ISA, qui n'existent pas dans le modèle relationnel. Le niveau d'imbrication des attributs sera adapté au niveau d'imbrication des champs et compté de la même façon. Les groupes sont par contre communs aux deux modèles. Les contraintes des groupes en relationnel sont toutefois plus nombreuses. Nous ajouterons la contrainte référentielle et la contrainte d'équivalence, désignées respectivement par **ref** et **equ**. La règle concernant les groupes devient :

pas de contrainte $<$ **identifiant primaire et secondaire** $<$ **contrainte référentielle** $<$ **équivalence, coexistence, at-least-one et exclusion**

Nous avons choisi de considérer les contraintes référentielles comme étant plus complexes que les identifiants pour deux raisons. La première est que ces contraintes sont moins courantes que les identifiants. La deuxième raison est qu'elles demandent, pour pouvoir être comprises, la lecture de deux types d'enregistrements, le référencé et le référençant. L'identifiant ne demande, quant à lui, la lecture que d'un type d'enregistrement. La contrainte référentielle est toutefois moins complexe que celle d'équivalence. En effet, cette dernière implique la référence mais également une contrainte d'existence sur les valeurs du type référencé. Finalement, nous avons placer la contrainte d'équivalence au même niveau que les contraintes d'existence. Ce choix est dû au fait que nous n'avons pu établir un ordre précis entre les contraintes d'existence et celle d'équivalence.

Ceci cloture la partie concernant la taille et la complexité des schémas. Les mesures présentées peuvent encore être développées. En effet, il existe d'autres contraintes peu utilisées, que nous n'avons pas présentées ici. Il existe également des cardinalités d'attributs qui peuvent être plus difficile à comprendre que d'autres. En plus de cela, il reste encore à étudier ces mesures pendant leurs utilisations. Il nous faut également étudier, par des études de cas, les valeurs de taille et de complexité des schémas.

[6] Cfr. 3.1.1.

3.4 Influence du schéma de la base de données sur les requêtes

Dans cette section, nous étudierons l'influence du schéma sur les requêtes SQL. Nous commencerons par voir comment représenter la complexité du code SQL et les limites de cette représentation. Nous verrons ensuite quels sont les éléments complexes et quels sont leurs liens avec le schéma conceptuel, et en particulier quels sont les rapports avec la taille et la complexité du schéma. La caractérisation de la complexité des requêtes est un atout en soi pour l'étude de la maintenabilité. Cet outil permettra entre autres d'identifier les éléments difficiles à interprêter au sein du code SQL.

3.4.1 Complexité des requêtes

La mesure de la complexité a toujours été un point clé dans la maintenabilité. Comme nous l'avons présenté au chapitre 2, le terme complexité regroupe de nombreux concepts. Nous ne verrons pas ici les problèmes de coûts en espace et en temps pour les requêtes SQL. Nous nous concentrerons sur la complexité structurelle, qui reste une façon objective d'évaluer la complexité d'une requête en terme de compréhension, et qui reste un indicateur important pour la maintenabilité. Il n'existe pas actuellement de modèle et mesure permettant d'évaluer cette complexité structurelle pour les requête SQL, alors que ceux-ci sont nombreux pour le langage procédural. Intuitivement, il est possible de développer deux approches distinctes. La première est de traduire les requêtes SQL en code procédural et ensuite d'appliquer des mesures du code procédural reconnues. La deuxième est de développer de nouvelles méthodes et mesures spécifiques au langage SQL. On peut déjà imaginer les problèmes engendrés par ces deux approches. Pour la première, il nous faudra tout d'abord définir un ensemble de règles de traduction de SQL vers JAVA, par exemple, qui sera ici le langage de destination. Le choix des règles se justifiera au travers de plusieurs exemples. Les requêtes seront traduites en suivant un schéma de lecture standard du code SQL. Le deuxième problème est l'application de mesures de complexité sur la traduction. Il nous faudra alors identifier quelles sont les structures SQL qui ont engendré les structures JAVA considérées comme complexes. En ce qui concerne la deuxième approche, celle-ci a l'avantage de partir de zéro et donc de permettre la définition de mesures adaptées à SQL. Malheureusement, cela demande un accord sur le choix de ce qui est complexe et de ce qui ne l'est pas. Ce choix dépend d'avis d'experts qui peuvent varier d'un expert à l'autre. Nous pensons que les deux méthodes peuvent être combinées. En conservant une relation d'ordre entre la complexité des traductions de SQL en JAVA et la complexité obtenue par la deuxième approche, cela correspond à une certaine validation pour les règles de traduction et pour les mesures définies dans la deuxième approche.

Nous allons développer ici la première approche. La traduction de SQL en JAVA va permettre d'obtenir un graphe de contrôle de flux, sur lequel nous pourrons appliquer les mesures de complexité. Intuitivement, on peut faire le rapprochement entre un prédicat SQL et un **if-then-else**. Pour rappel, les prédicats sont une structure du langage SQL. Ils permettent la comparaison en plusieurs expressions au moyen d'un opérateur. Cet opérateur peut effectuer une comparaison avec une valeur déterminée ou encore le **null** SQL. Il peut également vérifier des conditions plus complexes comme l'appartenance à une table, l'unicité, etc. Le résultat d'un prédicat peut être "vrai", "faux" ou "inconnu". Nous reviendrons plus en détails sur les types de prédicats dans la section 3.6.3. Malheureusement, il n'existe pas de règles permettant de mettre en place une telle représentation. Les deux requêtes suivantes illustrent ce que nous entendons ici par complexité de requêtes :

```
select *
from CLIENT
```

```
select *
from CLIENT
where LOCALITE = 'Namur'
```

Ces deux requêtes simples montrent bien qu'il existe différents degrés de complexité dans les accès aux données. Pour pouvoir évaluer la complexité structurelle, nous allons d'abord transformer les requêtes en méthodes JAVA. Une telle transformation demande la définition de règles formelles. Ces règles prendront en entrée une requête SQL et donneront comme résultat le code procédural équivalent. Ces règles de transformation pourront également être utilisées si un changement de plateforme est requis. Par exemple, si pour une application le SQL doit disparaître au profit d'un langage procédural.

Nous allons présenter, au travers d'exemples, la transformation d'une requête SQL en méthode JAVA. Cette transformation ne sera faite que dans un seul sens. Pour plus de facilités, nous avons considéré que les données sont disponibles sous formes de collection JAVA et accessibles partout dans le programme. Ces données peuvent être extraites facilement de la base de données et placées dans des objets JAVA. Nous ne considérons pas ici les systèmes d'index, présents dans les base de données et simplifiant considérablement certaines recherches.

Le premier exemple est la requête :

```
select *
from CLIENT
```

La méthode JAVA résultant de la transformation est :

```
Set resultats = new Set();
Iterator <Client>it = clients.iterator();
while(it.hasNext()){
    Client cli = it.next();
    resultats.add(cli);
}
return resultats;
```

Le **select** SQL est ici représenté par un **while** JAVA car on doit parcourir tous les objets de type **Client**. On pourrait penser que la méthode JAVA ci-dessus n'est pas correcte puisqu'on parcourt l'ensemble des clients pour les copier un à un dans l'ensemble-résultat. On aurait pu en effet copier la référence de **clients** dans **resultat**. L'argument principal de cette solution est le gain de performance. Mais elle est contraire à l'intuition, surtout dans le cas où la requête contient au moins un prédicat, comme nous le verrons dans l'exemple suivant. Nous n'avons donc pas ici pris en compte la performance du code JAVA.

Le deuxième exemple introduit les prédicats dans le langage SQL.

```
select *
from CLIENT
where numCli = 1234
```

La méthode JAVA résultant de la transformation est :

```
Set resultats = new Set();
Iterator <Client>it = clients.iterator();
while(it.hasNext()){
```

```
    Client cli = it.next();
    if(cli.numCli == 1234){
      resultats.add(cli);
    }
}
return resultats;
```

Si le **select** SQL représente un **while** JAVA, les prédicats SQL peuvent quant-à-eux, être représentés par des **if-then-else**. Cette règle correspond bien à l'intuition. Dans la requête SQL, un client ne sera dans le résultat que si son numéro de client (**numCli**) est égal à **1234**. On verifie donc d'abord la condition pour ensuite effectuer l'action, ce qui correspond bien à un **if-then-else** dans les langages procéduraux.

On observe que les traductions sont simples et directes. Mais plusieurs problèmes apparaissent lorsqu'on étend le langage SQL considéré. Le premier est celui des variables hôtes. Pour rappel, ces variables proviennent du code qui à fait appel au code SQL. Ce code appelant peut, par exemple, être écrit dans le langage COBOL. Les variables hôtes sont reconnues dans la définition du embedded-SQL sous le terme de **host variable**. Ce mécanisme est très utile, c'est pourquoi nous avons choisi de le prendre en compte. La requête suivante illustre comment peuvent être utilisées les variables hôtes.

```
select *
from CLIENT
where numCli = :numCli
```

La traduction de ces variables va se faire par l'ajout d'un paramètre dans la méthode JAVA. On ajoutera un paramètre pour chaque variable différente détectée. L'utilisation du paramètre se fera de la même façon que pour les valeurs déjà fixées dans la requête, comme par exemple, la valeur **1234** représentant un numéro de client dans l'exemple précédent. On observe également que les variables hôtes ne complexifient pas la structure de la méthode JAVA.

Nous allons maintenant développer la traduction de deux des constructions les plus importantes dans le langage SQL. Ces deux constructions sont la requête imbriquée et la jointure interne. Voyons d'abord le problème de la requête imbriquée au travers de l'exemple suivant :

```
select *
from ARTICLE
where NumArt in (select NumArt
                 from LigneCommande)
```

La méthode JAVA résultant de la transformation est :

```
Set resultats = new Set();
Iterator <Article>it1 = articles.iterator();
while(it1.hasNext()){
    Article art = it1.next();
    Iterator <LigneCommande>it2 = lignesCommandes.iterator();
    while(it2.hasNext()){
        LigneCommande lc = it2.next();
        if(lc.numArt == art.numArt){
          resultats.add(art);
        }
```

```
    }
}
return resultats;
```

Il existe deux façons d'interpréter une requête imbriquée. La première est une analyse descendante. On part alors du début de la requête jusqu'à la fin de celle-ci pour interpréter et comprendre la requête. La deuxième façon consiste à partir du plus bas niveau des requêtes imbriquées, donc de la fin de la requête, pour remonter ensuite vers les niveaux supérieurs, après avoir résolu chaque sous-requête. Dans les deux cas, la requête imbriquée sera traduite par un **while**. Selon ces deux lectures, il existe deux façons de transformer une requête SQL en une méthode JAVA. Dans la méthode JAVA ci-dessus, nous avons appliqué l'analyse descendante. Cela s'observe dans l'ordre des **while** dans la méthode. On commence en effet par parcourir les objets **Article**, et pour chacun d'entre-eux, nous parcourons les lignes des commandes. L'équivalent au prédicat SQL de la requête est encore un **if-then-else** qui est exécuté dans la deuxième boucle.

Nous allons maintenant traduire cette requête selon une approche ascendante. Pour ce faire, nous avons besoin de nouveaux objets JAVA qui vont contenir les résultats des requêtes imbriquées. Ces objets sont des ensembles qui contiennent généralement plusieurs valeurs d'un même attribut. Afin de ne pas complexifier le code JAVA, nous avons considéré les attributs tels que **numCli**, **numArt**, etc., comme étant des types primitifs. Dans certaines des méthodes JAVA qui vont suivre, nous les considérerons soit comme des types primitifs (exemple : **int**, **char**, etc.), soit comme des classes représentant ces types (exemple : **Integer**, **Character**, etc.). Dans le premier cas, nous utiliserons alors le syntagme **==** pour la comparaison. Dans le deuxième cas, nous utiliserons la méthode **equals**, ou encore **compareTo** pour les chaînes de caractères. Dans le second cas, nous pourrons également ajouter les objets dans des ensembles, ce qui n'est pas possible avec des types primitifs. La traduction est la suivante :

```
Set resultats = new Set();
Iterator <LigneCommande>it1 = lignesCommandes.iterator();
Set numarts = new Set();
while(it1.hasNext()){
    LigneCommande lc = it1.next();
    numarts.add(lc.numArt);
}
Iterator <Article>it2 = articles.iterator();
while(it2.hasNext()){
    Article art = it2.next();
    if(numarts.contains(art.numArt)){
      resultats.add(art);
    }
}
return resultats;
```

L'appel de méthode **numarts.contains(art.numArt)** traduit la lecture ascendante de la requête SQL au niveau du prédicat **in**. Cette méthode indique si un élément (dans l'exemple : **art.numArt**) se trouve dans un ensemble (dans l'exemple : **numarts**). Toutefois, une telle méthode peut ne pas être applicable dans le cas du langage JAVA si la méthode **equals**, testant l'égalité, n'est pas définie sur les valeurs des objets mais sur les références des objets. Si la traduction de la requête se fait dans un langage autre que JAVA, la méthode **contains** peut simplement ne pas exister. Les lignes de code suivantes :

```
if(numarts.contains(art.numArt)){
```

```
      resultats.add(art);
    }
```

doivent alors être remplacées par une construction similaire à l'exemple suivant :

```
Iterator <NumArt>it3 = numArt.iterator();
while(it3.hasNext()){
    NumArt na = it3.next();
    if(na.equals(art.numArt)){
      resultats.add(art);
    }
}
```

On constate dans cet exemple, que les deux méthodes n'ont pas une structure similaire par rapport aux **while** et aux **if-then-else**. L'approche ascendante génère ici un nombre de **while** et de **if-then-else** identique. Les graphes de contrôle de flux des deux versions (descendante et ascendate) auront donc le même nombre de noeuds de contrôle, mais n'auront pas la même structure. Il nous faut encore déterminer comment se comporte cette différence lorsque le nombre de sous-requêtes augmente ainsi que lorsque le type du prédicat SQL varie. Voyons tout d'abord comment peut se traduire une requête plus complexe à 3 niveaux. Cette requête donne les commandes dont au moins un des détails concerne le produit **PA60** et ce détail a une quantité commandée strictement inférieure à celle du détail relatif à la commande **30182** et qui concerne le produit **PA60**. On voit directement qu'une requête à trois niveaux devient plus complexe à comprendre et à formuler. Voici le code SQL de cette requête :

```
select *
from COMMANDE
where NCOM in (select NCOM
                 from DETAIL
                 where NPRO = 'PA60'
                 and QCOM < (select QCOM
                               from DETAIL
                               where NPRO = 'PA60'
                               and NCOM = '30182'));
```

Nous traduisons cette requête par la méthode suivante, en utilisant l'analyse descendante :

```
Set resultats = new Set();
Iterator <Commande>it1 = commandes.iterator();
while(it1.hasNext()){
    Commande com = it1.next();
    Iterator <Detail>it2 = details.iterator();
    while(it2.hasNext()){
        Detail det1 = it2.next();
        if(det1.ncom == com.ncom){
          if(det1.npro.compareTo("PA60")){
            Iterator <Detail>it3 = details.iterator();
            while(it3.hasNext()){
                Detail det2 = it3.next();
                if(det1.qcom < det2.qcom){
                  if(det2.npro.compareTo("PA60")){
```

```
                    if(det2.ncom == 30182){
                      resultats.add(com);
                    }
                  }
                }
              }
            }
          }
    }
}
return resultats;
```

Comme pour la requête précédente, le prédicat SQL **in** est évalué dans la sous-requête à laquelle il se rapporte. Ce prédicat aurait pu être évalué plus tard, par exemple dans le dernier **if-then-else** dans la méthode JAVA, sans changer le résultat final. Mais comme nous simulons ici une analyse descendante, celui-ci est évalué le plus tôt possible dans le code JAVA. Il en est de même dans la troisième boucle, dans laquelle on évalue en premier le prédicat **in** servant de liaison entre la première et la deuxième sous-requête. Observons maintenant la transformation de la requête selon la méthode ascendante :

```
Set resultats = new Set();
Iterator <Detail>it1 = details.iterator();
QCom qc;
while(it1.hasNext()){
    Detail det1 = it1.next();
    if(det1.npro.compareTo("PA60")){
      if(det1.ncom == 30182){
        qc = det1.qcom;
        break;
      }
    }
}
Iterator <Detail> it2 = details.iterator();
Set ncoms = new Set();
while(it2.hasNext()){
    Detail det2 = it2.next();
    if(det2.npro.compareTo("PA60")){
      if(det2.qcom < qc){
        ncoms.add(det2.ncom);
      }
    }
}
Iterator <Commande>it3 = commandes.iterator();
while(it3.hasNext()){
    Commande commande = it3.next();
    if(ncoms.contains(commande.ncom)){
      resultats.add(commande);
    }
}
```

La structure du code JAVA varie selon que l'on ait choisi une lecture descendante ou ascendante de la requête, que ce soit pour la répartition des noeuds **if-then-else**, que celle des noeuds **while**. Le nombre de noeuds de décision reste quant à lui identique pour les deux

versions. Nous généraliserons cette différence entre la lecture descendante et ascendate, en ce qui concerne la structure. Nous considérerons également comme vrai le fait que le nombre de noeuds est égal pour les deux lectures. Nous justifions cela au travers des deux exemples et par le fait que les prédicats SQL sont de deux types, ceux s'appliquant sur des ensembles, comme le prédicat **in**, et ceux s'appliquant sur des valeurs particulières, comme les prédicats **is null**, $<$, etc. Les deux exemples précédents couvrent ces deux types de prédicats.

Nous voyons également que le niveau d'imbrication des noeuds de décision dans l'approche ascendante est moins élevé que pour celui de l'approche descendante. Cela est dû au fait que la requête imbriquée est d'abord résolue et qu'elle donne un résultat qui n'est utilisé qu'après la sous-requête. Par la suite, nous ne traduirons plus les requêtes imbriquées que selon une lecture descendante, sauf mention contraire. Comme pour les requêtes simples et les prédicats, la transformation SQL vers JAVA des requêtes imbriquées reste simple et intuitive, en particulier pour l'approche descendante. Voyons maintenant comment considérer les jointures internes.

Afin de simuler la jointure interne, nous avons aussi besoin de nouveaux objets JAVA. Ces nouveaux objets vont contenir les résultats des jointures, c'est-à-dire le produit relationnel, parfois aussi appelé abusivement produit cartésien des tables de la jointure. Ces objets auront comme attributs tous les attributs de chaque table de la jointure. Nous retrouverons donc généralement, dans chacun de ces objets, des attributs redondants sur lesquels seront basés les conditions de jointure. La traduction de la requête en une méthode JAVA va correspondre à une interprétation générale des jointures internes. Nous commencerons par effectuer le produit relationnel des tables, pour ensuite vérifier les conditions de jointure et les autres prédicats. Bien entendu, cette démarche n'est pas appliquée par le SGBD car cette méthode serait alors trop coûteuse. Mais nous ne tiendrons pas compte ici des problèmes de performance. Afin d'illustrer la transformation d'une requête comprenant une jointure interne, nous utiliserons l'exemple suivant :

```
select NOM, NCOM, DATECOM
from CLIENT CLI, COMMANDE COM
where COM.NCLI = CLI.NCLI
and LOC = 'Toulouse'
```

Le produit relationnel de la jointure est d'abord exécuté dans la méthode JAVA et va donner tous les couples de **CLIENT** et de **COMMANDE** possibles. Il peut être représenté de la manière suivante :

```
Set <CommandeClient>commandesclients = new Set();
Iterator <Client>it1 = clients.iterator();
while(it1.hasNext()){
    Client cli = it1.next();
    Iterator <Commande> it2 = commandes.iterator();
    while(it2.hasNext()){
        Commande com = it2.next();
        CommandeClient comcli =
                new CommandeClient(cli.ncli,cli.nom,cli.adresse,
                    cli.localite,cli.cat,cli.compte,
                    com.ncom,com.datecom,com.ncli);
    }
}
```

Le produit relationnel est donc une source non-négligeable de complexité. En effet, pour chaque jointure interne, le nombre nécessaire de **while** imbriqués pour effectuer le

produit relationnel est égal au nombre de tables dans la jointure. Une fois le produit relationnel effectué, la méthode parcourt les objets nouvellement créés et vérifie les prédicats de la requête :

```
Set <ResultatComCli>resultats = new Set();
Iterator <CommandeClient>it1 = commandesclients.iterator();
while(it1.hasNext()){
    CommandeClient comcli = it1.next();
    Iterator <Commande> it2 = commandes.iterator();
    if(comcli.comncli == comcli.clincli){
      if(comcli.cliloc.compareTo("Toulouse")){
        ResultatComCli res =
                  new ResultatComCli(comcli.clinom,
                        comcli.comncom, comcli.comdatecom);
        resultats.add(res);
      }
    }
}
return resultats;
```

Une fois la jointure effectuée, la traduction des prédicats de la requête se fait facilement, comme on peut le constater dans le code JAVA ci-dessus. On observe également que, comme les éléments résultant du produit relationnel se trouvent dans l'ensemble **commandesclients**, il n'est plus nécessaire de parcourir les ensembles **clients** et **commandes**. La structure d'une jointure interne ne paraît donc pas à première vue beaucoup plus complexe que celle d'une requête utilisant des requêtes imbriquées. Nous ne pouvons affirmer pour l'instant qu'une requête utilisant la jointure est toujours plus ou moins complexe qu'une requête donnant un résultat équivalent et utilisant des requêtes imbriquées. Pour prouver cela, il nous faut d'abord déterminer quelles sont les interrogations de la base de données qui peuvent être exprimées pour un résultat équivalent sous la forme d'une jointure interne ou d'une requête imbriquée. Il faut ensuite pouvoir donner un ensemble de règles de transformations des requêtes imbriquées vers les jointures et inversement. Sur base de ces règles, nous pourrions ensuite prouver formellement lequel des procédés est le plus complexes en terme de structure. La définition de ces règles dépasse le cadre de ce mémoire, nous nous contenterons donc ici de l'étude de deux exemples. Le premier est la requête suivante utilisant des sous-requêtes :

```
select NCOM, DATECOM
from COMMANDE COM
where NCLI in (select NCLI
               from CLIENT
               where LOCALITE = 'Poitiers')
```

Cette requête est équivalente à la suivante qui utilise la jointure :

```
select NCOM, DATECOM
from CLIENT CLI, COMMANDE COM
where COM.NCLI = CLI.NCLI
and LOCALITE = 'Poitiers'
```

La première requête peut être traduite en JAVA, selon une lecture descendante, dans le code suivant :

```
Set resultats = new Set();
Iterator <Commande>it1 = commandes.iterator();
while(it1.hasNext()){
    Commande com = it1.next();
    Iterator <Client>it2 = clients.iterator();
    while(it2.hasNext()){
        Client cli = it2.next();
        if(com.ncli == cli.ncli){
          if(cli.localite.compareTo("Poitiers")){
            resultats.add(art);
          }
        }
    }
}
return resultats;
```

La deuxième requête qui utilise la jointure peut se traduire en JAVA de la façon suivante :

```
Set <CommandeClient>commandesclients = new Set();
Iterator <Client>it1 = clients.iterator();
while(it1.hasNext()){
    Client cli = it1.next();
    Iterator <Commande> it2 = commandes.iterator();
    while(it2.hasNext()){
        Commande com = it2.next();
        CommandeClient comcli =
                new CommandeClient(cli.ncli,cli.nom,cli.adresse,
                    cli.localite,cli.cat,cli.compte,
                    com.ncom,com.datecom,com.ncli);
    }
}

Set <ResultatComCli>resultats = new Set();
Iterator <CommandeClient>it1 = commandesclients.iterator();
while(it1.hasNext()){
    CommandeClient comcli = it1.next();
    Iterator <Commande> it2 = commandes.iterator();
    if(comcli.comncli == comcli.clincli){
      if(comcli.cliloc.compareTo("Poitiers")){
        ResultatCom res = new ResultatCom(comcli.comncom, comcli.comdatecom);
        resultats.add(res);
      }
    }
}
return resultats;
```

Dans cet exemple, nous pouvons observer que la première méthode JAVA comporte moins de points de contrôle (**while** et **if-then-else**) que la deuxième méthode. On constate également que l'imbrication des points de contrôle est plus importante dans la première méthode que dans la deuxième. Ces observations correspondent bien à l'intuition que nous avons en analysant les requêtes SQL. Le deuxième exemple confirme cette observation, que

nous considérerons comme généralisée par la suite. La requête utilisant des sous-requêtes est la suivante :

```
select *
from COMMANDE
where NCOM in (select NCOM
                  from DETAIL
                  where NPRO = 'PA60'
                  and QCOM < (select QCOM
                                 from DETAIL
                                 where NPRO = 'PA60'
                                 and NCOM = '30182'));
```

La requête équivalente utilisant une jointure est la suivante :

```
select M.NCOM, DATECOM, NCLI
from COMMANDE M, DETAIL D1, DETAIL D2
where M.NCOM = D1.NCOM
and D1.NPRO = 'PA60'
and D2.NCOM = '30182'
and D2.NPRO = 'PA60'
and D1.QCOM < D2.QCOM
```

Nous ne donnerons des méthodes JAVA que la structure des noeuds de contrôle. La figure 3.14 décrit cette structure. A gauche, se trouve la structure de la méthode correspondant aux requêtes imbriquées, lues de manière descendante. Au milieu, se trouve la structure de la première requête, lue de manière ascendante. A droite, se trouve la structure de la requête utilisant la jointure. On constate, de nouveau, que la requête avec requêtes imbriquées, comporte moins de noeuds de décision par rapport à la jointure. Cette différence, tout comme dans l'exemple précédent, est de 1 **while**. De même en ce qui concerne les niveaux d'imbriquation des noeuds de décision, la première structure possède une imbriquation plus élévée que la troisième. La deuxième structure est par contre moins imbriquée et contient moins de noeuds de décision que la jointure.

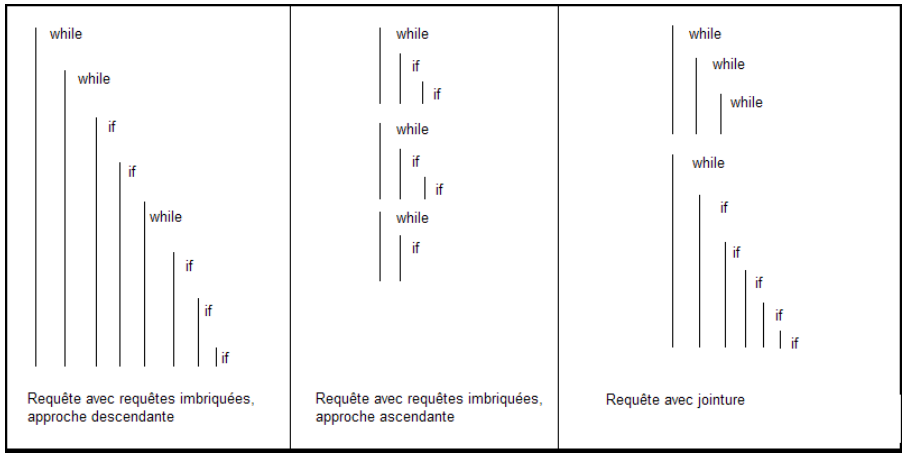

FIG. 3.14 – Structure générale des points de contrôle

La traduction des requêtes en méthode JAVA peut, naturellement, être encore développée car nous n'avons considéré ici qu'une partie très limitée du langage SQL. Nous

ne nous attarderons pas plus sur les règles de traduction des requêtes de type **select-from-where**. Nous allons maintenant aborder le problème des contraintes intégrées et gérées par le SGBD. Pour ce faire, nous utiliserons le mécanisme de suppression géré par le SGBD ainsi qu'une requête de suppression comme exemple. La gestion des éléments d'une table référençant une autre table, dont on supprime les éléments, est généralement traîtée de trois façons différentes. La première est le mode cascade qui va également supprimer les lignes de la table qui référencent l'élément supprimé. La construction SQL correspondante est **ON DELETE CASCADE**, qui est ajouté à la déclaration de la contrainte référentielle. La deuxième méthode est le changement de valeur de la clé étrangère. La valeur peut être mise à **NULL**, dans ce cas on ajoutera **ON DELETE SET NULL** à la contrainte référentielle. Elle peut également être mise à la valeur par défaut. La troisième façon de gérer la suppression est de ne rien faire et d'annuler la suppression lorsqu'une clé étrangère en mode bloquant référence la ligne supprimée. Ce type de mécanisme permet de remplacer un ensemble de requêtes de vérifications et de suppressions qui aurait pu être très lourd et très coûteux. Il serait donc intéressant de pouvoir les considérer dans notre analyse. De plus, ils représentent un élément à maintenir et ils "complexifient" la gestion des données.

Intuitivement, lorsqu'une contrainte référentielle est en mode cascade ou changement de valeur, nous nous représentons une lecture de la table concernée, une identification des éléments touchés et, finalement, la suppression de la ligne ou le changement de valeur de la clé étrangère. Ce schéma peut bien entendu être représenté sous la forme de code JAVA. Nous allons illustrer cela avec la requête suivante :

```
DELETE FROM COMMANDE
WHERE NCOM = '30182'
```

S'il existe un mode **ON DELETE CASCADE** sur la clé étrangère de la table **DETAIL** vers la table **COMMANDE**, une suppression de la commande **30182** entraînera d'abord la suppression de tous les détails qui référencent la commande. Le mode cascade doit être décrit explicitement dans la méthode JAVA afin de mettre en évidence la complexité globale de la requête. La suppression et le mode "cascade" peuvent être traduit comme suit :

```
Iterator <Commande>it1 = commandes.iterator();
while(it1.hasNext()){
    Commande com = it1.next();
    if(com.ncom == 30182){
      Iterator <Detail>it2 = details.iterator();
      while(it2.hasNext()){
          Detail det = it2.next();
          if(det.ncom == com.ncom){
            details.remove(det);
          }
    }
    commandes.remove(com);
}
return;
```

La prise en compte du mode "cascade" a, ici, doublé le nombre de noeuds de décision. En toute généralité, le mode "cascade" ajoutera une boucle et un nombre de **if-then-else** égal au nombre d'attributs constituant la clé étrangère. Comme pour les exemples précédents, cette méthode JAVA ne tient pas compte des problèmes d'optimisation. Il est bien entendu possible de réorganiser le code afin de simuler un mécanisme de type "trigger" lors de la suppression d'une commande. Mais ce que nous avons souhaité mettre en évidence ici, est le raisonnement suivi lors de la lecture du code. Lors d'un processus de

maintenance, la personne qui doit analyser le code doit prendre en compte tous les aspects de ce dernier. Il doit donc raisonner sur le code visible, les requêtes, mais aussi sur les mécanismes cachés, comme les modes ”cascade”, les checks, ou encore les triggers. En exprimant le code déclaratif sous une forme procédurale, nous pouvons donc bien utiliser les méthodes et mesures concues pour ce dernier. Nous ne détaillerons pas plus les règles de traduction qui sont encore nombreuses, ni les autres mécanismes SQL, tels que les triggers. Enfin, il existe un aspect pratique sur lequel cette traduction fait défaut. Il est possible de construire les requêtes dynamiquement. Ces constructions dynamiques peuvent changer plus que les valeurs et paramètres des requêtes, elles peuvent modifier la structure, par exemple, en rajoutant des prédicats. Dans ce cas, il devient difficile d'énumérer toutes les requêtes du système.

3.4.2 Influence du schéma sur les requêtes

Nous abordons, brièvement, dans cette partie, le problème de la relation entre les schémas des données et les requêtes ainsi que les autres mécanismes SQL. Nous présenterons nos intuitions sur ce lien et quelques pistes concernant l'étude pratique de cette relation.

Intuitivement, lorsque la partie base de données est bien construite et utilisée par le système, nous savons qu'il existe un lien entre le schéma et les constructions SQL. Il est normal que les tables qui proviennent des types d'entités soient lues, à moment ou à un autre, au moyen d'une requête. Il est également logique que des données soient insérées dans la base. Ces exemples sont constituent des évidences mais il existe des cas plus particuliers.

On peut se demander s'il existe un lien entre les chaînes de contraintes référentielles du schéma et le nombre de requêtes utilisant des sous-requêtes et des jointures dans l'application. En effet, leur existence est généralement due à de telles contraintes. Ces dernières proviennent, quant à elles, des relations de type un-à-plusieurs du schéma conceptuel. La longueur des chaînes de références influence peut-être aussi le niveau d'imbrication des sous-requêtes. Les requêtes SQL ne sont pas les seules constructions dont nous pouvons parler. Les mécanismes tels que les triggers peuvent implémenter des contraintes spécifiques au domaine d'application mais aussi des contraintes d'existence. Or, ces dernières peuvent être dues aux relations ISA. Il peut donc exister un lien entre la présence de certaines contraintes dans le schéma et le nombre ainsi que la complexité des triggers. Il est aussi possible que le nombre de prédicats présents dans les requêtes dépende du nombre de champs et d'attributs des schémas. Un plus grand nombre de champs peut signifier un plus grand nombre de critères de recherches et donc un plus grand nombre de requêtes différentes.

Ces éléments peuvent difficilement être étudiés de manière théorique. Pour cette raison, nous allons donner ici une liste d'éléments à étudier au travers d'études de cas. Ces études devraient mettre en évidence la relation entre le schéma et les accès aux données ou encore montrer que celle-ci est très faible.

Nous pensons que la relation entre certains éléments des schémas et les constructions SQL est de type linéaire. Cette hypothèse sera bien entendu à vérifier sur des études de cas. Dans les relations linéaires, nous devons choisir les variables dépendantes et les variables indépendantes. Parmi les variables dépendantes, nous identifions le nombre de requêtes, la complexité des requêtes, le nombre de jointures et de sous-requêtes, le niveau d'imbrication des sous-requêtes, le nombre de prédicats concernant une valeur ou une variable et, finalement, le nombre de prédicats sur une sous-requête. Dans ces éléments, nous avons parlé de la complexité des requêtes. Celle-ci pourra être évaluée par transformations des requêtes en code procédural et ensuite par l'application d'une méthode telle que celles définies par McCabe [39] et Fenton [15].

Les variables indépendantes qui devront être prises en compte sont les suivantes : La taille des schémas conceptuels et logiques, le nombre et le type des éléments complexes identifiés en 3.3.2. Par exemple, nous étudierons la taille, le nombre et l'arité des types d'associations, le nombre, la structure et la contrainte des groupes, le nombre de relation ISA, l'imbrication moyenne des attributs, etc.

Les éléments choisis peuvent, en général, être facilement identifiés dans les schémas et les requêtes. Cela facilitera l'étude pratique. Il faudra toutefois porter attention au type de logiciels choisis. En effet, certains paramètres pourront dépendre de la classe du logiciel, comme c'est le cas dans [43].

3.5 Maintenabilité du logiciel

Cette section présente les aspects de la maintenabilité du logiciel dans son ensemble, influencés par le schéma des données et les accès aux données. Nous allons présenter deux problèmes distincts. Le premier est celui de la cohésion du logiciel. La deuxième concerne l'évaluation de la taille fonctionnelle du logiciel, à partir du schéma et des accès aux données.

3.5.1 Cohésion du logiciel

La cohésion est un attribut du logiciel utilisé lors d'une assertion de la maintenabilité. Plus un module est cohérent, plus celui-ci sera maintenable. Il existe plusieurs types de cohésions mais celle qui va être retenue ici concerne les données utilisées par les différentes parties du programme. Si on suppose un système utilisant une base de données, une partie ou l'ensemble de ses modules vont avoir accès à la base de données. Des données seront alors échangées entre les deux systèmes. La cohésion des données peut être présentée comme suit : deux modules distincts doivent avoir accès à des tables ou des entités différentes et une table ou une entité ne doit être en interaction qu'avec un seul module.

Ce principe entraîne une cohésion au niveau du traitement des données.

L'étude de la cohésion sous l'angle des données permet de mettre en place une mesure et un modèle simples et automatisables. Les entrées qui nous intéressent ici sont les différents modules du logiciel pouvant être identifiés par leur nom, les accès à la base de données qu'ils produisent et la structure de la base de données. L'étude de ces trois éléments permet pour chaque module de définir un ensemble de tables utilisées par le module. Le processus d'identification de ces tables, à partir d'une analyse des requêtes, sera illustré dans la section consacrée à la taille du logiciel. Une bonne cohésion suggère qu'il y ait le moins d'intersections possibles entre ces ensembles de tables.

Soit n, le nombre de modules du système, $T_1, \ldots, T_n$, les ensembles de tables associés à chaque module et $t_1, \ldots, t_m$ l'ensemble des tables de la base de données alors,

- si $\forall\, i :\ 1 \leq i \leq m : \#\{j : 1 \leq j \leq n : t_i \in T_j\} \leq LimiteMin \Rightarrow$ la cohésion étudiée à partir des accès aux données est maximale. Il n'existe pas de problèmes de cohésion de par l'utilisation de ces tables ;
- si $\forall\, i :\ 1 \leq i \leq m : \#\{j : 1 \leq j \leq n : t_i \in T_j\} \geq LimiteMax \Rightarrow$ les tables répondant à ces critères sont des tables techniques. Il ne s'agit pas d'un problème de cohésion mais d'une contrainte technique ;
- si $\forall\, i :\ 1 \leq i \leq m : LimiteMin < \#\{j : 1 \leq j \leq n : t_i \in T_j\} < LimiteMax \Rightarrow$ il existe peut-être un problème de cohésion au niveau de t_i et des modules M_j. Le problème potentiel ne pourra être confirmé ou infirmer que par une

étude plus approfondie.

Ce modèle rudimentaire permet d'identifier les tables et modules "à problèmes" au travers de comparaisons avec une borne inférieure et une borne supérieure. Ces deux bornes peuvent être exprimées en nombre de modules ou, moyennant une légère modification de la formule, en pourcentage par rapport au nombre total de modules. La borne inférieure permet de définir à partir de combien de modules ayant accès à une même table le système manque de cohésion par rapport à cette table. La deuxième borne permet de mettre en évidence les tables particulières de la base de données qui, pour des raisons techniques, sont utilisées par la majorité des modules. Ces tables, appelées tables techniques, sont, en général, des tables contenant des paramètres globaux pour l'ensemble du système ou encore des tables "journal" dans lesquelles sont écrits les événements du système.

Cette mesure de la cohésion utilise donc des attributs internes et se rapporte à l'analysibilité et la variabilité dans le modèle ISO9126. Ces sous-caractéristiques sont étudiées de manière interne et le modèle défini peut intervenir durant la phase d'écriture du code et après celle-ci. Il est également possible de définir une mesure similaire pour les phases de design à condition que les documents de conception montrent les flux d'informations entre la base de données et les modules. La mesure et le modèle sont objectifs. Les résultats ne changent pas selon l'environnement. Mais ce modèle demande un ajustement des $LimiteMin$ et $LimiteMax$ basées sur une étude empirique.

Par rapport à l'échelle de cohésion de Yourdon et Constantine [47], cette mesure de la cohésion ne donne des informations que sur un aspect de la cohésion communicationnelle. Elle ne permet donc pas de déterminer si un système possède une cohésion plus élevée que la cohésion communicationnelle mais permet déterminer si le logiciel ne possède pas ce niveau de qualité. Enfin, nous avons remarqué que la mise en pratique de cette mesure peut être difficile dans le cas où tous les accès à la base de données se font au travers d'un seul module. Il faut alors identifier ce module, déterminer quelles sont les requêtes et associer chaque requête au module qui l'a déclenchée.

3.6 Taille du logiciel

La taille du logiciel est un facteur indépendant de la maintenabilité du logiciel. Toutefois, la taille d'un logiciel est déterminante lorsqu'il s'agit d'évaluer les coûts d'un processus d'ingénierie d'un logiciel. Ces processus sont par exemple la conception, les tests, la maintenance, etc. La taille est donc un des attributs les plus importants du logiciel. De plus différents travaux ont prouvé qu'il existe un lien entre la taille du logiciel et la taille et la structure des données. Ces travaux ont été présentés dans les sections 2.2.2 et 2.2.3. Cette première raison nous amène à étudier également l'influence des données sur le logiciel en terme de taille.

Une deuxième raison, plus concrète, nous a amené à étudier cette influence. Il s'agit de l'application pratique établie durant le stage lié à ce mémoire. Durant ce stage, une estimation automatisée de la taille, en points de fonction, faite à partir de la structure des données et de leurs accès a pu être développée. Cette application sera présentée dans le chapitre 4 consacré à la mise en place des techniques et outils développés. Mais celle-ci nous a permis de développer une théorie plus générale que nous allons présenter ici. Il s'agit donc ici d'une estimation par rétro-ingénierie du code de l'application. Ce problème a été introduit dans [4].

Cette partie commencera par une description brève d'une méthode permettant de calculer la taille d'une requête. Nous verrons ensuite les nouveaux principes permettant de décrire une requête de façon à reprendre les éléments communs aux mesures de taille fonctionnelle.

3.6.1 Approche générale pour estimer la taille d'une requête SQL

La taille d'une requête est une caractéristique qui reste assez peu abordée. Il est toutefois possible de voir une requête en terme de nombre d'éléments lus et écrits dans les tables de la base de données. Ces lectures et écritures sont proches des mouvements de données. Le mouvement de données est un concept qui est, dans cette partie, repris de la méthode COSMIC-FFP [1]. Le principes des mouvements de données se retrouve dans la plupart des mesures de taille fonctionnelle. La méthode développée ici permet d'estimer la taille fonctionnelle d'une application, c'est-à-dire d'estimer le nombre de points de fonction. Les mouvements de données sont de quatres types différents. Ceux-ci sont les entrées, les sorties, les lectures et les écritures. Ces mouvements de données, dans la méthode COSMIC-FFP, concernent des groupes de données qui sont, en général, les entités du schéma conceptuel du logiciel.

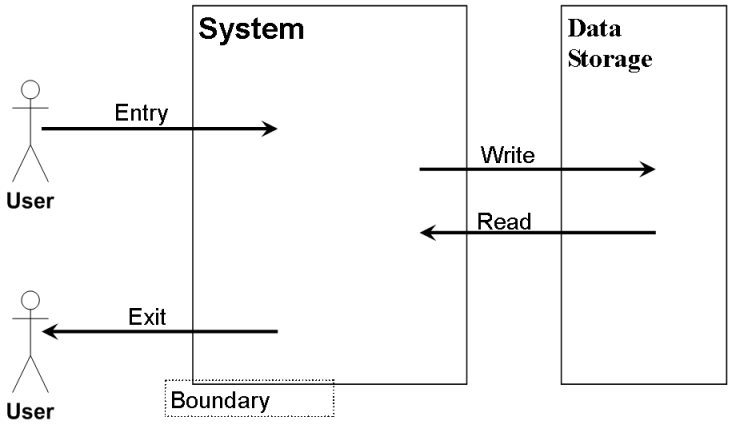

FIG. 3.15 – Vue du système selon la mesure COSMIC-FFP

La figure 3.15 donne une vue d'ensemble sur la façon dont la méthode COSMIC-FFP représente le système. Cette vue prend en compte l'entièreté du système, mais il est possible de l'adapter aux accès aux données et plus particulièrement aux requêtes SQL, comme cela a été développé durant le stage. En utilisant le principe des mouvements de données, il est alors possible d'établir la taille d'une requête. On peut par exemple estimer la lecture et l'écriture dans une table, effectuée dans la requête, comme ayant chacun un score de 1. Ce calcul simple permet dès lors d'obtenir une taille de type fonctionnel pour les requêtes. Ce résultat n'apporte malheureusement pas d'informations réellement utiles par rapport à l'ensemble du logiciel. Les points que nous venons de développer peuvent par contre être interprétés d'une façon plus générale. En effet, il est possible d'intégrer la taille " fonctionnelle " d'une requête dans l'ensemble de l'application. La figure 3.16 montre la nouvelle façon dont le système peut être vu.

Cette nouvelle interprétation va être développée dans le reste de ce chapitre.

3.6.2 Interprétations des accès aux données

Le point précédent a présenté le fait que les mouvements de données constituent un concept proche des requêtes SQL. Une estimation de la taille de l'application peut être obtenue à partir des requêtes, en adaptant le concept des points de fonctions à ces requêtes.

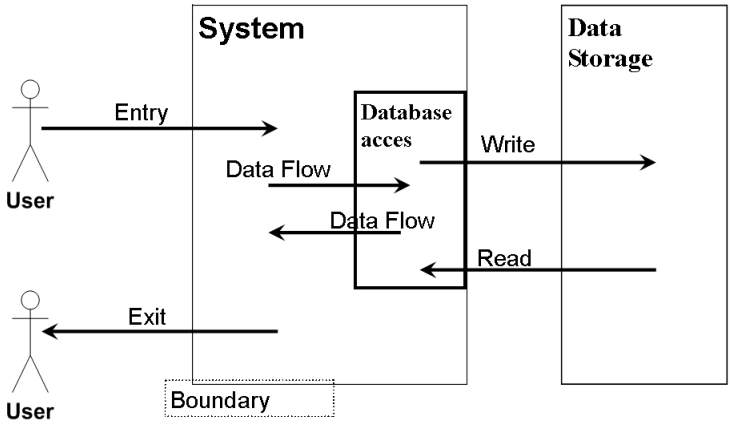

FIG. 3.16 – Mise en évidence des accès aux données

Pour ce faire, nous posons les hypothèses suivantes :

- l'application considérée utilise une base de données pour le stockage et la consultation des éléments représentés dans le schéma conceptuel des données ;
- les fonctions utilisateurs proposées par l'application utilisent les données situées dans la base de données ;
- l'application effectue des manipulations simples, ne nécessitant pas d'algorithme complexe[7].

Ces hypothèses nous mettent en présence d'un système d'informations. Ce type de logiciel représente une grande partie de la population des logiciels. Elles nous permettent également de dériver les éléments suivants :

- une fonction utilisateur de l'application, contiendra au moins une instruction SQL de type lecture, suppression, mise à jour ou insertion ;
- l'instruction SQL sera composée de mouvements de données, qui feront partie de l'ensemble des mouvements de données de la fonction utilisateur à laquelle l'instruction SQL se rapporte ;
- l'instruction SQL recevra les données directement transmises par l'utilisateur légèrement[8] ou non modifiées ;
- l'instruction transmettra à l'utilisateur des données légèrement ou non modifiées.

En partant de ces deux groupes d'hypothèses, il est possible d'estimer la taille fonctionnelle d'une application à partir de ses accès aux données. Il n'existe pas de démonstrations de ces hypothèses. Bien entendu, le premier groupe d'hypothèses montre une vision " utopique " d'un système d'informations et il n'existe que peu de systèmes respectant entièrement ces principes. Malgré tout, nous avons supposé que la plupart des systèmes d'informations restaient assez proches du type de logiciels que nous venons de présenter [9]. La méthode que nous allons développer dans la section suivante va donc nous permettre d'estimer la taille fonctionnelle de ces applications à partir de la structure physique des données et des accès SQL.

[7] Il s'agit là d'une hypothèse provenant de la mesure de taille fonctionnelle Cosmic-FFP et qui fait référence aux algorithmes tels ceux du traitement d'images, de videos, ceux utilisés dans les domaines scientifiques avancés, etc.

[8] Une modification sera considérée comme légère si elle permet encore de reconnaître l'attribut et la table auxquels correspond la donnée.

[9] Cette raison fait que nous parlons d'estimation et non pas d'un calcul exact de la taille fonctionnelle.

3.6.3 Réécriture générale des accès

Les deux sections précédentes ont introduit l'idée générale d'une estimation de la taille fonctionnelle d'un logiciel à partir des requêtes SQL. COSMIC-FFP a été utilisé pour présenter cette nouvelle méthode. Nous avons également remarqué que les principes abordés dans les deux sections précédentes sont généralisables à d'autres mesures de taille fonctionnelle que COSMIC-FFP. En effet, même si une des particularités de COSMIC-FFP est sa simplicité [10], ses types de mouvements de données se retrouvent définis de manière plus générale dans les autres mesures de points de fonctions. La base théorique des points de fonction [3] s'appuie sur ces principes au travers des types de points de fonction suivants :

- inputs externes ;
- outputs externes ;
- fichiers logiques internes ;
- fichiers externes d'interface ;
- recherches externes.

Ces catégories sont détaillées dans la plupart des mesures de taille fonctionnelle. Elles représentent la façon dont se déroule l'échange d'informations entre un système et ses utilisateurs. Bien qu'associés à certains types d'échanges particuliers, ces points de fonction utilisent les principes d'entrée, de sortie, de lecture et d'écriture que nous avons vu précédemment.

Une différence observée entre les mesures de taille fonctionnelle est la façon dont elles considèrent l'unité d'information. Par exemple, un attribut d'une entité constitue la plus petite unité de mesure dans la méthode développée par le groupe IFPUG [21]. L'unité de mesure est par contre l'entité dans la méthode COSMIC-FFP. Il est donc intéressant de pouvoir définir une requête SQL de manière plus abstraite de façon à faire ressortir les éléments qui nous intéressent pour une telle analyse.

Si nous partons de la requête simple [11] et de la définition de l'entité client à la figure 3.17 :

```
select *
  from CLIENT
    where NCLI = :numCli
```

Les éléments qui vont nous intéresser dans cette requête sont :

- **select ***
- **from CLIENT**
- **where NCLI = :numCli**

select * :

Cette construction constitue le retour d'informations de la requête et donc la sortie de la requête, que nous allons considérer comme étant l'information fournie à l'utilisateur. Cette information n'a fait l'objet que de modifications simples entre la sortie de la requête et sa lecture par l'utilisateur. On peut donc supposer que les éléments constituant la sortie de la requête seront de même nature que ceux lus par l'utilisateur. Ces éléments appartiennent

[10] Cette simplicité provient du fait que la méthode COSMIC-FFP ne distingue pas plusieurs types de fonctions utilisateur comme le font les autres mesures de taille fonctionnelle. Ce choix constitue une facilité lors de son utilisation mais entraine également un manque de précision.

[11] La requête est écrite en SQL92/EMBEDDED. L'élément " numCli " est donc une variable fournie par l'application et contenant la valeur d'un numéro de client (ncli).

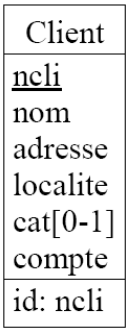

FIG. 3.17 – Schéma de l'entité Client.

à une ou plusieurs entités bien déterminées du schéma conceptuel de l'application. Dans l'exemple ci-dessus, tous les attributs de l'entité **Client** et donc l'entité **Client** elle-même constitue la sortie de la requête.

Selon la mesure de points de fonction utilisée, les informations sont comptées différemment. Par exemple, la norme COSMIC-FFP compte les mouvements de données en nombre d'entités. Les mesures de taille fonctionnelle plus proche du modèle d'Albrecht comptent les informations en termes de champs d'informations simples, c'est-à-dire généralement en termes d'attributs d'entités. Afin de garder une représentation de la requête utilisable pour la plupart des mesures de taille fonctionnelle, nous avons choisi de réécrire **select *** de la façon suivante :

" EXIT : (CLIENT.NCLI, CLIENT.NOM, CLIENT.ADRESSE, CLIENT.LOCALITE, CLIENT.CAT, CLIENT.COMPTE) "

De cette façon, toutes les informations nécessaires à l'analyse de la requête en terme de taille fonctionnelle sont présentes dans cette représentation.

Une première règle simple peut déjà être donnée par rapport à ces résultats :

Les sorties d'une requête sont tous les attributs présents dans la clause " select " de la requête.

from CLIENT :

Cette clause de la requête SQL correspond à la lecture de l'entité **Client**. En effet, **from CLIENT** indique que la table lue et utilisée par la requête est la table **CLIENT**. La clause **from** peut bien entendu être plus complexe et peut également appartenir à des requêtes SQL autres que les **select-from-where**. Nous supposons bien entendu que toutes les tables lues par la requête le sont par intention et non par erreur lors de l'implémentation des requêtes. Il existe des constructions plus complexes. En effet, le langage SQL permet d'appeler des tables contenues directement dans la base de données ou construites au travers d'une autre requête. Le langage permet également de faire appel aux jointures externes et internes, afin de construire de nouvelles tables, généralement temporaires ou persistantes au travers d'une vue. Ces constructions de tables peuvent être utilisées dans la clause **from** de la requête. L'estimation des lectures dans la requête devient alors plus complexe. Nous illustrerons ces constructions et leurs analyses par la suite.

En ce qui concerne la représentation des lectures de la requête, celle-ci peut être réécrite de la façon suivante :

" READ : (CLIENT.NCLI, CLIENT.NOM, CLIENT.ADRESSE, CLIENT.LOCALITE, CLIENT.CAT, CLIENT.COMPTE) "

Cette représentation permet d'abstraire le langage SQL tout en gardant les informations sur les éléments lus par la requête. De nouveau, nous pouvons définir une règle simple sur base de cet exemple :

Les lectures d'une requête sont tous les attributs de toutes les tables appelées dans la clause from de la requête.

where NCLI = :numCli :

Cette construction correspond à une entrée, c'est-à-dire une donnée fournie par l'utilisateur. La façon dont est transmise cette donnée n'a pas d'importance. L'entrée peut être par exemple saisie au clavier, faire partie d'une liste, etc. **NCLI = :numCli** est donc un prédicat de comparaison en SQL. En sachant que l'information transmise par l'utilisateur est **:numCli**, on observe facilement qu'elle correspond à l'attribut **NCLI** de la table **CLIENT** et donc de l'entité **Client**.

Comme nous l'avons dit précédemment, les prédicats SQL ne sont pas toujours aussi simples. Dans les différents types de prédicats, il est possible d'avoir plusieurs attributs et plusieurs informations provenant de l'utilisateur pour un même prédicat. Aussi, avant de développer des prédicats plus complexes, il nous faut faire des choix quant à la façon dont les entrées vont être analysées. Nous allons donc définir deux principes afin de résoudre ces problèmes. Ceux-ci peuvent paraître arbitraires et mènent dans certains cas à des résultats inexacts, mais ils nous ont permis de garder le caractère " automatisable " de l'analyse. Ces deux principes sont les principes de décompositions des prédicats et d'association des variables. Ils ont été définis pour la méthode COSMIC-FFP mais sont, bien entendu, applicables à une analyse indépendante de la mesure de taille fonctionnelle utilisée.

Le problème est donc de pouvoir associer une variable avec un groupe d'attributs appartenant à une ou plusieurs entités. Il faut pouvoir identifier les attributs auxquels se rapportent les variables utilisées dans les requêtes SQL.

Le premier principe est celui de la décomposition des prédicats SQL. Il se base sur les prédicats suivants :

- comparison predicate ;
- between predicate ;
- in predicate ;
- like predicate ;
- null predicate ;
- quantified comparaison predicate ;
- exists predicate ;
- unique predicate.

Il permet également de décomposer les types SQL appelés **CaseExp**. Ce type regroupe quatre types d'expressions "case", deux expressions abrégées et deux expressions standards.

Ces prédicats et ces "cases" peuvent être décomposés en un ou plusieurs couples. Chaque couple est une association symbolisant une comparaison entre le membre de gauche et celui de droite. Par exemple, la comparaison simple (comparison predicate) fait appel à un membre de gauche, un membre de droite et un opérateur de comparaison. Cet opérateur n'a pas d'importance dans l'analyse, ce qui fait que le prédicat peut être redéfini sous forme d'une simple association entre le membre de gauche et celui de droite. Les prédicats plus complexes comme le **between**, se basent également sur des comparaisons.

Par exemple, **X between Y and Z** est équivalent à **Y <= X and X <= Z**. Ces deux comparaisons peuvent alors être redéfinies en deux associations (X,Y) et (X,Z) car Y et Z n'ont aucune relation entre eux.

Ce principe permet donc de transformer l'ensemble des prédicats d'une requête SQL en un ensemble de couples, et permet d'appliquer le principe d'association des variables. Il en est de même pour les expressions SQL de type "case", celles-ci peuvent être redéfinies sous forme d'association. Nous utiliserons l'expression "case" suivante comme exemple : **nullif(5, :var)**. Cette expression est équivalente à : **case when 5 = :var then null else 5**. On constate donc bien la présence de comparaison au sein des **CaseExpr**. Exprimer les comparaisons sous cette forme entraîne une perte d'informations sur la requête, mais l'information perdue n'est pas nécessaire à l'estimation des entrées.

Par contre, les prédicats de type **unique**, **null** et **exists** ne permettent pas de lier une variable à une ou plusieurs tables car la comparaison qu'ils effectuent se fait sur un élément fixe et pas sur un groupe de données/une entité, quel que soit le prédicat.

La décomposition des prédicats permet de simplifier le problème mais ne nous permet pas de lier une variable à un ou plusieurs attributs. Le deuxième principe permet donc d'associer une variable d'une partie gauche ou droite d'une association avec le membre opposé de l'association. Il ne s'agit que d'une estimation puisque la variable est associée à l'ensemble des attributs du membre opposé ainsi qu'a l'ensemble des sorties des sous requêtes directes du membre opposé. Cette approximation permet d'automatiser le comptage des entrées car sans celle-ci, il reviendrait à un expert le choix du lien entre une variable et une ou plusieurs entités.

A l'aide de ces deux principes, il est possible de représenter **NCLI = :numCli** de la façon suivante :

" ENTRY : ((:numCli,(CLIENT.NCLI))) "

Il n'est malheureusement pas possible de donner une règle aussi simple que les deux règles précédentes en ce qui concerne les entrées. Toutefois si nous gardons à l'esprit la décomposition des prédicats et l'association des variables, il est possible de définir la règle suivante :

Les entrées de la requête sont tous les attributs présents dans la partie opposée à celle contenant une variable fournie par l'utilisateur. Cette partie opposée est un des deux membres d'un couple obtenu par la représentation des prédicats et " cases " selon les principes de décomposition des prédicats.

Ceci clôture l'analyse de notre premier exemple. Cette première présentation ne permet pas de prendre en compte toutes les constructions utilisables dans les requêtes du langage SQL. Nous allons donc détailler l'analyse de certaines constructions les plus fréquemment utilisées.

Jointures et clés étrangères :

Nous allons tout d'abord aborder la représentation des jointures et des clés étrangères au travers de la requête suivante :

```
select Article.noArticle, coalesce(sum(quantite),0) Quantite
from Article left join LigneCommande
on LigneCommande.noArticle=Article.noArticle
```

```
group by Article.noArticle
```

Les entités sont définies de le schéma conceptuel, à la figure 3.18.

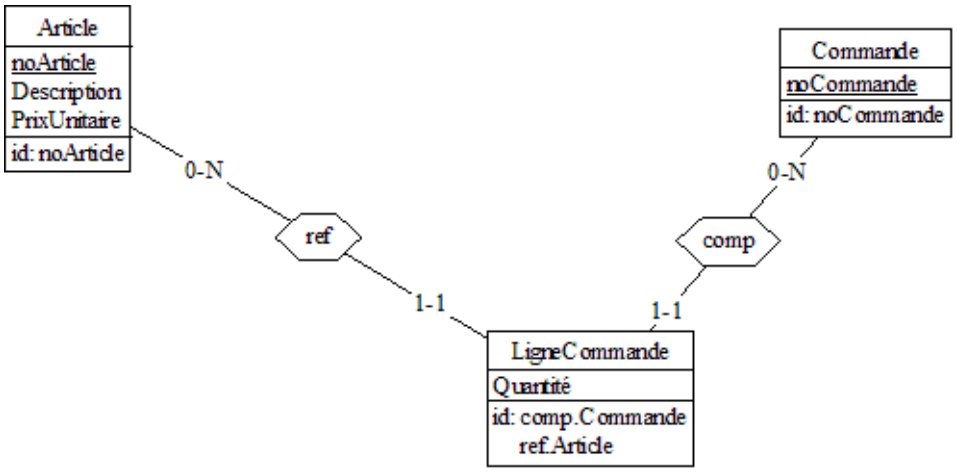

FIG. 3.18 – Schéma des entités Article, LigneCommande et Commande

Le schéma ci-dessus peut se traduire directement sous sa forme physique. Les deux associations **ref** et **comp** se retrouvent donc sous forme de clés étrangères dans la base de données. On peut dès lors retrouver facilement le schéma conceptuel à partir du schéma physique des tables.

La requête ci-dessus est composée de trois clauses. Celles-ci sont les clauses **select**, **from** et **group by**. Cette dernière ne nous intéresse pas dans notre analyse car elle ne modifie ni les lectures, ni les entrées, ni les sorties.

La clause **select** peut être représentée de la façon suivante :

" EXIT : (ARTICLE.NOARTICLE, LIGNECOMMANDE.QUANTITE) "

La présence de **ARTICLE.NOARTICLE** est évidente. En effet, l'attribut est présenté explicitement dans la clause. La construction **coalesce(sum(quantite),0)** demande par contre une analyse plus poussée. La fonction coalesce est en réalité une expression " case " du langage SQL et donne comme résultat le premier paramètre différent de **null** de la fonction. Il peut donc s'agir soit de **sum(quantite)** soit de **0**. **sum(quantite)** correspond à une somme des quantités de la table **LIGNECOMMANDE**. Le traitement effectué par la fonction **sum** ne modifie pas la nature de l'attribut qui fait l'objet de la somme. Pour cette raison, nous le retrouvons directement dans la représentation de la clause. **0** n'est quant à lui pas présent car il s'agit d'une valeur arbitraire n'appartenant à aucune table et donc entité.

Nous pouvons définir de façon plus formelle la représentation des sorties :

La sortie d'une requête correspond à la sortie de la clause select de la requête. La sortie de cette clause contient tous les attributs se trouvant dans la clause à l'exception de ceux se trouvant dans un case ou dans une sous - requête. La sortie de la clause select contient aussi la sortie de toutes les sous-requêtes directement présentes dans la clause select. Finalement, la sortie de la clause select contient également tous les attributs pouvant être le résultat d'un " case " contenus dans la

clause. Dans le cas où le résultat d'un " case " contiendrait une sous-requête ou un autre case, les règles à appliquer sont les mêmes que celles définies pour la sortie d'une clause select.

La clause **from** peut être représentée de la façon suivante :

" READ : (ARTICLE.NOARTICLE, ARTICLE.DESCRIPTION, ARTICLE.PRIXUNITAIRE, LIGNECOMMANDE.QUANTITE, ARTICLE.NOARTICLE, COMMANDE.NOCOMMANDE) "

La représentation ci-dessus nous montre la façon dont sont interprétées les clés étrangères. Un attribut ou un groupe d'attributs référençant une table restent des éléments de la table référencée. Pour cette raison, **ARTICLE.NOARTICLE** est présent deux fois dans la représentation des lectures. De même, la référence de **LIGNECOMMANDE** vers **COMMANDE** a impliqué dans cette requête la présence de **COMMANDE.NOCOMMANDE** dans les lectures.

On observe dans cet exemple que la présence d'une jointure externe ne présente pas de problème particulier. En effet, l'utilisation d'une jointure externe ou interne dans la clause **from** correspond à la lecture des deux tables de la jointure.

Nous avons pu également illustrer la façon dont peuvent être représentées les clés étrangères. Cette représentation semble être la forme la plus minimale. Toutefois, il pourrait être intéressant de garder de l'information sur toute la référence, c'est-à-dire la table référencée, l'attribut ou le groupe d'attributs constituant la référence et la table qui référence. Ce choix pourra être discuté par la suite.

Nous pouvons formaliser l'interprétation des clés étrangères de la façon suivante :

L'attribut ou le groupe d'attributs présents dans une table et constituant une clé étrangère doit être représenté comme appartenant toujours à la table référencée.

La règle concernant les lectures des requêtes ne doit pas être modifiée. En effet, la règle actuelle tient compte des tables appelées dans la clause **from** de manière générale. C'est-à-dire tant sous leur forme la plus courante, que dans leur utilisation au sein d'une jointure.

Requêtes imbriquées :

Voici maintenant une dernière requête illustrant la façon dont sont représentées les requêtes contenant une requête ou plusieurs requêtes imbriquées. L'exemple est le suivant :

```
select unique (Art.noArticle)
from Article Art
where Art.noArticle in (select LigneCommande.noArticle
                            from LigneCommande)
```

Le schéma conceptuel correspondant aux deux tables utilisées dans la requête est le même que celui de la requête précédente.

La représentation des sorties de la requête ne pose pas de problème et est construite de la même façon que pour les exemples précédents. Étant donné l'absence de variables, il n'y a pas d'entrées dans cette requête. Nous nous intéresserons donc aux lectures. La requête comporte deux clauses **from**. Ces deux clauses contiennent les tables lues dans la requête. Nous pouvons donc établir la règle suivante :

Les lectures d'une requête SQL sont constituées de tous les attributs se trouvant dans les tables déclarées de chaque clause from de la requête.

Cet exemple termine notre développement sur la réécriture des accès aux données au travers des requêtes SQL. D'autres règles sont bien entendues encore à définir avant de pouvoir prendre en compte d'une part, la représentation des écritures et d'autre part, l'ensemble du langage SQL utilisable dans les requêtes. Le but de ce développement était principalement de présenter la logique de notre raisonnement au lecteur. Une application concrète sera présentée dans le chapitre suivant. La définition sera appliquée dans le cadre de la méthode COSMIC-FFP.

3.6.4 Utilisation de cette représentation pour estimer la taille du logiciel

La représentation abstraite des requêtes que nous venons de définir n'a pas comme seul but de pouvoir donner les entrées, lectures, etc., des requêtes SQL. En effet, comme nous l'avons dit précédemment, nous cherchons ici à pouvoir estimer la taille fonctionnelle du logiciel uniquement à partir d'un schéma des données et des accès SQL du logiciel. Les mesures de taille fonctionnelle étant nombreuses et utilisant différentes méthodes, la représentation abstraite des requêtes a pour but de faciliter l'application de ces mesures sur des informations préétablies. En effet, les règles que nous venons de donner mettent en évidence les éléments nécessaires aux différentes mesures. Chaque mesure devra alors être définie de façon précise. Leur définition devra pouvoir utiliser les informations contenues dans la représentation de la requête.

Une fois la mesure de taille fonctionnelle formalisée et mise en place, l'analyse des représentations des requêtes SQL pourra avoir lieu. Le résultat de cette analyse constituera alors une estimation de la taille fonctionnelle du logiciel. Selon le degré de ressemblance du système par rapport à celui que nous avons défini dans la section 3.6.2, l'estimation se rapprochera plus ou moins de la taille fonctionnelle réelle du système. Certains cas particuliers seront bien entendu à prendre en compte selon la mesure utilisée. Par exemple, on pourra choisir de ne pas considérer les tables techniques. Dans ce cas, il reviendra à un expert de déterminer quelles sont ces tables[12]. Certaines mesures ne considèrent également pas les informations redondantes, c'est-à-dire les attributs de même type se trouvant dans les lectures, les entrées, les écritures ou encore les sorties. Ces règles seront à mettre en place lors de l'implémentation de la mesure. Elles permettront de rapprocher au mieux la valeur de l'estimation de la valeur réelle du système.

L'utilité du calcul de la taille fonctionnelle à partir d'un logiciel déjà implémenté réside dans le fait que la taille fonctionnelle du logiciel peut ne pas être connue. Cette information peut être nécessaire dans certains processus liés au logiciel, généralement afin de calculer le coût d'une opération. La taille fonctionnelle ne peut actuellement être calculée précisément par rétro-ingénierie qu'avec l'aide d'un expert. Ce calcul se révèle donc très coûteux et prend plus ou moins de temps selon la taille du système, de la documentation disponible et de la qualité du code. L'automatisation des mesures de taille fonctionnelle reste encore un problème actuel. La méthode que nous venons de mettre en oeuvre ne permet pas d'obtenir des résultats précis. Pour cette raison nous parlons uniquement d'estimation. Mais une estimation reste intéressante en particulier dans le cas ou la méthode est rapide et automatisée.

Nous présenterons dans le chapitre suivant une implémentation de la mesure COSMIC-FFP ainsi qu'une application de celle-ci sur une partie d'un logiciel particulier.

[12] L'identification de ces tables pourra également se faire au moyen de la mesure de la cohésion, qui identifie les tables techniques, définie au point 3.5.1.

Chapitre 4

Applications des outils développés

Le chapitre précédent a décrit de manière théorique les mesures et modèles envisageables à partir du schéma des données et des requêtes à la base de données. Ce chapitre présente la définition concrète et l'application de deux des techniques présentées dans le chapitre précédent. La première concerne la taille du logiciel et plus précisément sa taille fonctionnelle. La deuxième technique abordée dans ce chapitre concerne la complexité des requêtes.

4.1 Mesure de la taille

Nous avons décrit dans le chapitre précédent une représentation abstraite des requêtes. Cette représentation avait pour but de synthétiser la requête afin de ne garder que les éléments utiles à une estimation de la taille fonctionnelle du logiciel à partir des requêtes SQL. Nous allons maintenant mettre en pratique cette représentation ainsi que l'estimation de la taille au travers de la méthode COSMIC-FFP [1]. Cette partie commencera donc par une introduction à la méthode et mesure de taille fonctionnelle COSMIC-FFP. Nous présenterons ensuite la façon dont les requêtes SQL et la mesure peuvent être associées afin d'estimer la taille fonctionnelle des requêtes. Ce résultat correspondra à la taille fonctionnelle du logiciel telle que nous l'avons présenté dans le chapitre précédent. Nous appliquerons ensuite la méthode à un logiciel. Finalement, nous étudierons les résultats obtenus et présenterons quelques avantages et limites de cette méthode.

4.1.1 Introduction à Cosmic-FFP

La mesure Cosmic-FFP [1] [18] permet de calculer la taille fonctionnelle d'un système. L'application de la méthode Cosmic-FFP demande en premier lieu une délimitation du système, c'est-à-dire une identification de la frontière. Une fois la frontière délimitée, il faut pouvoir identifier les groupes de données et les processus fonctionnels.

Les groupes de données sont généralement des types d'entités identifiés dans le schéma conceptuel des données du système, qu'on appelle également schéma de la statique. Les informations les plus utiles provenant du schéma sont les types d'entités, leurs attributs et les relations entre les types d'entités. Les processus fonctionnels sont les " Functional User Requirements " ou encore FUR, c'est-à-dire les fonctions utilisateur requises. Ces fonctions sont généralement décrites sous forme de use cases au début de la conception du système. Elles sont situées à l'intérieur de la frontière du système mais interagissent obligatoirement avec au moins un utilisateur. L'utilisateur peut être un être humain ou un autre système.

Une fois les groupes de données et les FUR identifiés, il faut calculer la taille fonctionnelle de chaque processus fonctionnel. La taille fonctionnelle de tout le système est obtenue en additionnant la taille de chaque processus du système. La taille fonctionnelle d'un processus est calculée sur base des entrées, sorties, lectures et écritures effectuées dans le processus. Ces entrées, sorties, lectures et écritures sont appelées mouvements de données, il en existe donc 4 types dans la méthode COSMIC-FFP. Chacun de ces mouvements de données décrit la façon dont un groupe de données voyage dans le système.

Une entrée correspond à un groupe ou une partie d'un groupe de données fourni par l'utilisateur du système. Une sortie correspond à un groupe de données allant du système vers l'utilisateur. Une lecture correspond à une entrée d'un groupe de données provenant de l'espace des données, par exemple une base de données. Une écriture est une écriture d'un groupe de données par le système dans l'espace des données. Le schéma ci-dessous présenté dans le chapitre précédent synthétise ces éléments.

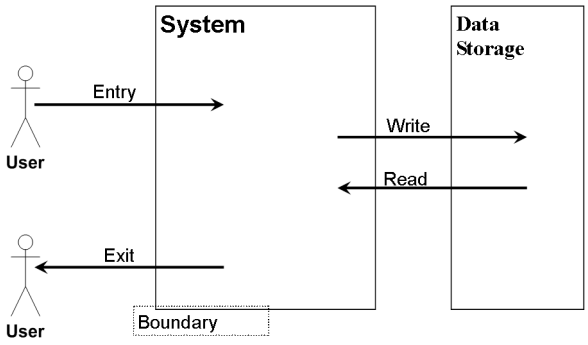

FIG. 4.1 – Vue du système selon la mesure COSMIC-FFP

La méthode COSMIC-FFP se distingue également des autres mesures de taille fonctionnelle par d'autres propriétés. Tout d'abord, les mouvements de données sont comptés en types d'entités et non en attributs. Un mouvement de données ne concernant qu'un attribut d'un type d'entités sera donc vu comme un mouvement de données relatif à tout le type d'entités. Ce manque de granularité constitue un des points critiqués de la méthode mais la rend également plus simple. Une autre propriété de la mesure COSMIC-FFP est que les mouvements de données redondants ne sont pas comptabilisés. Contrairement à la mesure définie par Albrecht et aux autres mesures s'en inspirant directement, COSMIC-FFP ne distingue pas plusieurs classes de fonctions utilisateur. Toutefois 4 grands types de fonctions peuvent être donnés, ceux-ci étant les vues, les mises-à-jour, les insertions et les suppressions. L'appartenance d'un processus utilisateur à un de ces 4 types ne modifie pas la façon dont sont comptés les mouvements de données.

Il existe évidemment d'autres règles particulières à la méthode que nous ne décrirons pas ici. Celles-ci concernent, par exemple, l'identification des couches, des sous-systèmes ou encore les règles particulières de redondance. Nous signalons également que COSMIC-FFP n'est pas adapté pour le calcul de la taille des systèmes basés sur des algorithmes lourds et complexes ainsi que des systèmes utilisant le traitement de variables en continu, c'est-à-dire l'audio et la vidéo.[1]

[1] Ces types de logiciels demandent un effort d'implémentation important pour un nombre limité de fonctionnalités utilisateur. La mesure Cosmic-FFP n'est donc pas appropriée pour ce type de logiciels.

4.1.2 Mapping entre les deux concepts

Concepts communs et limites

Avant d'adapter la méthode décrite au chapitre précédent, nous avons procédé à une mise en correspondance entre les principes de Cosmic-FFP et le langage SQL. Celle-ci a permis d'identifier les liens généraux entre ces deux concepts et a également mis en évidence les premières limites de l'automatisation. Les premières limites identifiées concernent l'identification du système, de sa frontière, des groupes de données et des processus fonctionnels.

Nous avons tout d'abord constaté que l'estimation de la taille fonctionnelle d'un système au travers de ses accès aux bases de données ne permet pas d'identifier la frontière de ce système. Cette identification revient donc à un expert. Ce fait n'est toutefois pas dommageable car les requêtes se trouvent à l'intérieur du système. Cependant, dans le cas où une application serait composée de plusieurs systèmes, comme par exemple une application client-serveur, l'analyse de l'ensemble de l'application donnerait lieu à l'analyse de deux systèmes distincts ayant chacun leur frontière. Ce qui signifierait que l'estimation de toute l'application donnerait comme résultat la somme de l'estimation du client et du serveur. En pratique, l'identification des frontières n'est pas une opération complexe et celle-ci peut même être faite manuellement sans l'aide d'un expert. De même, les couches composant le système ne peuvent être identifiées à partir des requêtes SQL.

La méthode que nous utilisons se base sur le back-end du système/de l'application, sur ses accès à la base de données. Cela ne permet donc pas de parcourir l'ensemble du processus fonctionnel en partant du front-end, c'est-à-dire les interfaces utilisateurs, et de tracer les mécanismes du processus. Une telle analyse a été considérée au début de la partie pratique du mémoire mais a été écartée étant donné sa complexité. Nous ne pouvons donc pas identifier un processus fonctionnel au sens Cosmic-FFP mais uniquement faire la supposition qu'une requête est déclenchée par un processus fonctionnel et qu'un processus fonctionnel ne contient qu'une requête.

Cette supposition sera mise en doute dans les sections suivantes. Nous montrerons qu'il existe des différences entre les concepts de COSMIC-FFP et ceux utilisés pour l'automatisation. Ces différences sont principalement dues au fait qu'un système d'information réel diverge de celui que nous avons présenté au chapitre précédent. Malgré l'impossibilité de reconnaître exactement un processus, nous pouvons néanmoins donner les éléments suivants :

- Tout processus fonctionnel effectue une lecture et cette lecture passe, selon le domaine d'application de la méthode, par un accès à la base de données ;
- Tout comme les lectures, les écritures passent également par la base de données ;
- La traduction d'une requête est très similaire à un processus. On peut donc supposer qu'une partie des sorties du processus, sinon la totalité, sont des entités se trouvant dans la base de données sous forme de tables et qui vont être extraites grâce à une requête ;
- Ce dernier élément est également valable pour les entrées fournies par l'utilisateur, qui sont en général des contraintes de recherche ou des nouvelles valeurs dans la requête.

Tout comme pour les divergences entre COSMIC-FFP et les requêtes SQL, nous pouvons établir les liens entre les deux concepts.

Tout d'abord, les groupes de données sont généralement les entités présentes dans un schéma conceptuel, un diagramme UML, etc., représentant la statique du système. Ils peuvent également être identifiés par un expert au travers des use cases du système. Dans le cas de l'automatisation, le fait de se baser sur le langage SQL nous a obligé à nous

baser sur le schéma physique de la base de données. Par exemple, les relations binaires ayant les cardinalités de leurs rôles valant respectivement 1-1 et 1-N ne sont pas reconnues dans le schéma physique, mais doivent être gérées d'une autre façon comme par exemple à l'aide des triggers. Un autre exemple de différence est le changement de nom. En effet, les noms des entités et de leurs attributs peuvent être changés durant la transformation du conceptuel au physique, à cause de contraintes imposées par le SGBD. Il existe donc des différences importantes entre ces deux niveaux mais afin de limiter le problème et de ne pas étendre la méthode, nous avons supposé que le schéma physique et le schéma conceptuel ne comportent pas de différences significatives, c'est-à-dire que le schéma conceptuel peut être retrouvé à partir du schéma physique, sans ambiguïté. Cela nous amène à une correspondance entre les entités au sens Cosmic-FFP et les tables de la base de données.

Comme nous en avons parlé dans la partie présentant la mesure Cosmic-FFP, il existe des mouvements de données. Ces mouvements sont à la base du résultat d'une estimation de la taille fonctionnelle puisque l'estimation de la taille du système dépend de leur occurrence. On remarque qu'une requête SQL effectue des lectures, des écritures, etc., et on retrouve donc au sein d'une requête les mouvements de données de Cosmic-FFP. Cette correspondance est certainement le point le plus important pour l'automatisation.

Description pratique de COSMIC-FFP

Comme le montre la description de Cosmic-FFP, une estimation fait appel à un ensemble de règles allant de la plus simple à la plus complexe, ces règles ne sont toutefois pas détaillées dans le manuel de mesure. Cette partie présente donc les règles de Cosmic-FFP considérées dans la méthode et les décrit au travers d'exemples. Ces définitions ont principalement pour but de décrire les mécanismes Cosmic-FFP en pratique et la façon dont ils sont interprétés dans la méthode d'automatisation. Dans ces définitions, nous utiliserons le terme "groupe de données" et "entité" de manière équivalente. Pour illustrer ces mécanismes, nous nous baserons sur quatre processus fonctionnels classiques. Ceux-ci sont la vue, l'ajout, la suppression et la modification. Ces processus condensent les interactions possibles entre les utilisateurs et un système, et coïncident avec les quatre types principaux de requêtes SQL qui sont les create, read, update et delete. Le schéma ci-dessous sera utilisé à titre d'exemple pour illustrer la structure des données sur lesquelles agissent les quatre processus [2].

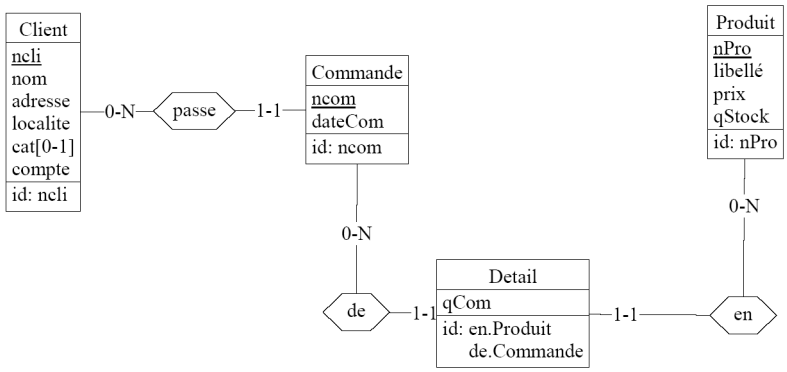

FIG. 4.2 – Schéma conceptuel.

[2]Ce schéma reste assez simple afin de ne pas aborder des problèmes qui n'ont pas été intégré dans la méthode d'automatisation comme par exemple les relations binaires aux cardinalités 1-1,1-N. Les relations plus complexes seront présentées dans le chapitre consacré aux problèmes identifiés.

Cas 1 : Une seule entité :

Pour ce cas, nous considérerons que le type d'entités **Client** est le seul connu du système. De ce fait, il n'existe donc plus de relation entre **Commande** et **Client**.

FIG. 4.3 – Schéma conceptuel.

Vue : Une vue du groupe de données **Client** donnera une lecture, une entrée et une sortie de **Client**. La lecture et la sortie sont directement sous-entendues dans une vue puisqu'il faut en effet lire les données de **Client** et les fournir à l'utilisateur comme résultat du processus. La prise en compte d'une entrée est due au fait que, comme cela est décrit dans le chapitre introduisant Cosmic-FFP, un processus n'est fonctionnel que s'il y a une action, donc une entrée, provenant de l'utilisateur. Cette contrainte sur les processus sera adaptée par la suite. Cette entrée permettra le plus souvent d'identifier l'occurrence ou les occurrences du type d'entités **Client** qui doivent se trouver dans la sortie du processus. Une autre sortie doit être comptée mais n'appartient pas au groupe de donnée **Client**. Cette sortie correspond aux messages d'erreurs, qui sont automatiquement comptés dans l'analyse de tout processus. Ces messages d'erreurs doivent donc être pris en compte pour les processus de suppression, d'ajout et de modification.

Suppression : La suppression d'un groupe de données aura comme résultat une lecture, une écriture, une entrée et une sortie (le message d'erreur). L'entrée permet, comme l'entrée de la vue, d'identifier les occurrences concernées. L'écriture est triviale puisqu'il s'agit d'un processus de suppression. La lecture apparaît car avant de supprimer l'entité, on vérifie si l'entité à supprimer existe bien.

Ajout : L'ajout fournit le même résultat que la suppression. La lecture s'effectue toutefois afin de vérifier que l'occurrence ou les occurrences de l'entité **Client**, qui vont être ajoutées, n'existent pas déjà dans le système de stockage.

Modification : Le processus de modification de l'entité **Client** donne les mêmes résultats que la suppression et l'ajout. On voit que la modification rassemble les principes de la suppression d'une occurrence (ou de plusieurs) et de la création d'une nouvelle qui aura le plus souvent des caractéristiques communes avec l'ancienne. Ce mélange entre l'ajout et la suppression fait que le mouvement de données de lecture correspond à une vérification de l'existence de l'ancienne version de l'entité et de la non-existence de la nouvelle version.[3]

Cas 2 : Deux entités liées par une relation :

Ce schéma inclut une relation entre deux entités. Cette relation sera traduite en clé étrangère au moment de la traduction du schéma dans la base de données.

[3]On constate qu'un processus fonctionnel peut ne pas s'effectuer complètement et donc, en pratique, ne pas utiliser certains mouvements de données. Cosmic-FFP ne considère pas un cas particulier du processus mais bien l'ensemble de ses possibilités. C'est pourquoi on estime par exemple la vue ci-dessus à 4 FFP, une lecture, une entrée et deux sorties.

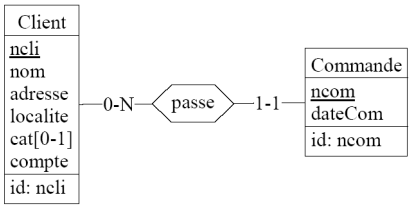

FIG. 4.4 – Schéma conceptuel

Vue de **Client** : Ce processus donne le même résultat que dans le cas 1.

Vue de **Commande** : Le résultat de cette vue sera influencé par les parties de **Commande** utilisées dans le processus. Le fait d'être une vue implique toujours des lectures, des entrées et des sorties mais les groupes de données se trouvant dans le résultat seront influencés par la façon dont est construit le processus. Nous allons tout d'abord détailler les lectures. La vue lit toute l'entité **Commande**, elle lit les attributs de **Commande**, qui sont ncom et dateCom, et la relation que **Commande** a avec **Client**. La lecture de cette relation correspond à la lecture de l'entité **Client**. La vue aura donc deux groupes de données lus. Pour ce qui est des entrées et sorties, les entités concernées dépendent du fait que le mouvement de données concerne soit les attributs de **Commande**, soit la relation entre **Client** et **Commande**, ou encore les deux. Les résultats seront donc respectivement soit **Commande**, soit **Client** soit **Client** et **Commande**. Par exemple, dans le schéma ci-dessus, il est possible de trouver une **Commande** uniquement avec ncom car ncom est identifiant, dans ce cas, l'entrée de la vue est **Client**. Il est aussi possible de trouver les occurrences de **Commande** liées à un certains **Client** où **Client** est l'entrée de ce processus.

Suppression de **Client** : La logique de la mesure Cosmic-FFP diffère ici de la logique de base de données. En effet, la suppression de l'entité **Client** n'effectue pas de mouvement de données sur l'entité **Commande**. Il y a donc une différence importante entre les deux théories puisque, en théorie des bases de données, étant donné la relation **passe**, qui correspond à une contrainte référentielle, une suppression d'une occurrence entraîne au moins une lecture sur **Commande** et peut même donner lieu à une écriture. Ce fait n'est pas considéré dans les règles de mesures de Cosmic-FFP. Ce processus ne donnera donc qu'une lecture, une entrée, une écriture de **Client** ainsi qu'une sortie (le message d'erreur), c'est-à-dire le même résultat que dans le cas 1.

Suppression de **Commande** : La relation entre **Client** et **Commande** intervient ici aussi. Au niveau des entrées et lectures, l'analyse est semblable à celle de la vue de **Commande**. Pour les écritures, la suppression entraîne une écriture sur les attributs de **Commande** et sur la relation **passe**. Il y a donc deux écritures, une sur **Commande** et une sur **Client**.

Ajout de **Client** : Le résultat est identique à l'ajout dans le cas 1.

Ajout de **Commande** : Les règles utilisées pour l'analyse de ce processus sont semblables à celles vues lors de la lecture et la suppression. On a donc deux lectures, vérifiant que les occurrences de l'entité **Commande** à ajouter n'existent pas déjà. Puis on compte deux entrées qui correspondent aux données à écrire dans le processus et qui sont fournies par l'utilisateur. Ces données sont **Commande**, pour ses attributs, et **Client**, pour la relation entre les deux groupes de données. Ensuite, nous avons deux écritures, représentant les données écrites par le processus et finalement, nous avons une sortie correspondant au message d'erreur.

Modification de **Client** : Le raisonnement utilisé pour l'estimation de ce processus est semblable à celui de la suppression de **Client**. Il ne sera donc pas détaillé ici. Les

mouvements de données seront une lecture, une écriture, une entrée et une sortie.

Modification de **Commande** : De la même façon que pour le cas 1, la modification utilise les mécanismes de l'ajout et de la suppression. Le résultat de l'estimation est donc de deux lectures, **Client** et **Commande**, une entrée **Client** ou **Commande** ou deux entrées **Client** et **Commande** en fonction des champs modifiés et des occurrences de **Commande** qui doivent être modifiées. Les écritures varieront en fonction des attributs modifiés et/ou de la modification de la relation **passe** et seront soit **Client** soit **Commande** ou encore les deux. La sortie du processus sera un message d'erreur.

Les mécanismes présentés ci-dessus restent assez simples mais couvrent déjà une grande partie des cas possibles. On discerne bien le lien entre un processus et une requête SQL, et la façon dont Cosmic-FFP peut être associé aux accès aux bases de données. Ce sont ces associations qui vont être présentées dans les points suivant et qui constituent le corps de la méthode d'automatisation décrite dans ce mémoire.

4.1.3 Règles formelles et implémentations

Nous allons ici illustrer une partie du mapping entre la mesure Cosmic-FFP et les requêtes SQL. Le mapping a été traduit en règles formelles, basées sur la logique des prédicats. Ces règles ont ensuite été implémentées dans l'outils ASF+SDF [44]. L'ensemble des règles formelles utilisées se trouve dans l'annexe A.

Décomposition des prédicats et association des variables

Ces deux paragraphes rappellent les principes de décomposition des prédicats et d'association des variables, donnés au chapitre précédent.

Pour pouvoir automatiser l'estimation FFP à partir de requête SQL, deux principes ont été définis et utilisés dans la méthode. Le premier est le principe de décomposition des prédicats. Il permet de décomposer n'importe quel prédicat SQL en un ensemble d'associations. Chacune de ces associations représente une comparaison effectuée dans le prédicat. Une association est représentée sous la forme d'un couple de deux éléments, chacun de ces éléments étant une expression SQL pouvant être complexe. L'ensemble des associations ne permet toutefois pas de retrouver le type de comparaison utilisée par le prédicat mais celui-ci n'a pas d'influence dans la méthode Cosmic-FFP.

Le deuxième principe est celui de l'association des variables qui permet d'associer une variable provenant du langage ayant appelé SQL avec un ensemble de tables. Cette variable doit se trouver dans une des deux partie d'une association. L'ensemble des tables est obtenu à partir de l'autre partie de l'association. La règle permettant de définir cet ensemble de tables est donnée ci-dessous.

Définitions générales

Cette partie définit de manière générale les termes et types utilisés dans les parties suivantes.

- $table \equiv$ *table SQL*
- $TableDef \equiv$ *ensemble de tables avec leurs attributs respectifs*
- $Read \equiv$ *lecture au sens COSMIC-FFP*
- $Write \equiv$ *écriture au sens COSMIC-FFP*
- $Entry \equiv$ *entrée au sens COSMIC-FFP*
- $Exit \equiv$ *sortie au sens COSMIC-FFP*

- $query \equiv$ *requête Select-From-Where*
- $predicat \equiv$ *"équation" booléenne au sens SQL, utilisant les opérateurs du langage*
- $attribut \equiv$ *attribut SQL*
- $variable \equiv$ *paramètre d'une requête SQL, provenant du langage appelant la requête.*
- $association \equiv$ *couple représentant une partie ou l'ensemble d'un prédicat SQL et symbolisant une comparaison entre deux membres d'un prédicat*
- $EnsTables \equiv \{t_1, \ldots, t_n\}$, *un ensemble de tables*
- $EnsPredicats \equiv \{p_1, \ldots, p_n\}$, *un ensemble de prédicats*
- $EnsAssociations \equiv \{assoc_1, \ldots, assoc_n\}$, *un ensemble d'associations*

Fonctions prédéfinies

Cette partie présente les fonctions utilisées par les fonctions présentant de manière théorique l'estimation COSMIC-FFP. Les fonctions ci-dessous sont considérées comme données.

- $attribut(a, t) \equiv$ *vrai si a est un attribut de l'entité t, faux sinon. L'évaluation se fait à partir de TableDef*
- $query(r) \equiv$ *vrai si la requête est de type Select-From-Where, faux sinon*
- $clauseFrom(r) \equiv$ *renvoit la clause From d'une requête SQL*
- $predicat(r) \equiv$ *donne l'ensemble des prédicats d'une requête*
- $association(p) \equiv$ *donne l'ensemble des associations composant le prédicat p*
- $gauche(assoc) \equiv$ *donne la partie gauche de l'association assoc*
- $droite(assoc) \equiv$ *donne la partie droite de l'association assoc*

Règles de comptage

Cette section présente de manière formelle, l'estimation du comptage COSMIC-FFP sur une partie du langage SQL/embedded. La partie considérée reprend les requêtes de type **select-from-where** simples.

Fonction générale :

$$
\begin{aligned}
FFP(r) \quad = \quad & (Read + Entry + Write + Exit) \\
& \mid (r \in \{query\} \\
& \wedge table(r) \in def\_table \\
& \wedge Read \ = \ \#RTable(r) \\
& \wedge Write \ = \ \#WTable(r) \\
& \wedge Entry \ = \ \#ETable(r) \\
& \wedge Exit \ = \ \#ExTable(r)
\end{aligned}
$$

Comptage des lectures :

$$
\begin{aligned}
& RTable(r) \equiv \\
& \quad (query(r) \Rightarrow RTable(r) = (RFrom(r)) \\
& RFom(r) \equiv \\
& \quad \exists \{t_1, \ldots, t_n\} \, E \ : \\
& \quad (\forall \, table \, t_i \in E, (t_i \in clauseFrom(r) \\
& \quad \wedge t_i \in DefTable)) \\
& \quad \wedge (\nexists \, t \ : \ ((t \in clauseFrom(r) \\
& \quad \wedge t \in DefTable \wedge t \notin E) \\
& \quad \Rightarrow RFrom(r) \ = \ E
\end{aligned}
$$

Comptage des sorties :

$$
\begin{array}{l}
ExTable(r) \equiv \\
\quad (query(r) \Rightarrow \\
\quad\ \forall\, attribut\ a \in clauseSelect(r) : \\
\quad\quad \exists\, table\ t : attribut(a,t) \\
\quad\quad \wedge\, t \in DefTable \\
\quad\quad \Rightarrow ExTableA = \cup\{t\}) \\
\quad \Rightarrow ExTable(r) = ExTableA
\end{array}
$$

Comptage des écritures :

$$
\begin{array}{l}
WTable(r) \equiv \\
\quad (query(r) \Rightarrow WTable(r) = \emptyset)
\end{array}
$$

Comptage des entrées :

$$
\begin{array}{l}
ETable(r) \equiv \\
\quad (query(r) \Rightarrow ETable(r) = EPredicat(r) \\
EPredicat(r) \equiv \\
\quad ((\forall\, predicat\ p_i \in \{p_1, \ldots, p_n\}, \\
\quad\quad \{p_1, \ldots, p_n\} = predicat(r), \\
\quad \exists\, E_i,\ E_i = \{t1, \ldots, t_n\} : \\
\quad\quad \forall\, association\ assoc_j \in \{assoc_1, \ldots, assoc_n\}, \\
\quad\quad\ \{assoc_1, \ldots, assoc_n\} = association(p), \\
\quad\quad EAssoc(assoc_j) \subseteq E_i) \\
\quad \wedge (\nexists\, table\ t \in E_i \wedge t \in DefTable : \\
\quad\quad \forall\, association\ assoc_j \in \{assoc_1, \ldots, assoc_n\}, \\
\quad\quad\ \{assoc_1, \ldots, assoc_n\} = association(p), \\
\quad\quad t \notin \cup EAssoc(assoc_j))) \\
\quad \Rightarrow ETable(r) = \cup E_i \\
EAssoc(assoc) \equiv \\
\quad \exists\, \{t1, \ldots, t_n\}\ E : \\
\quad \forall\, table\ t \in E,\ t \in DefTable \wedge \\
\quad\ ((\exists\, variable\ v \in variable(gauche(assoc))) \\
\quad\quad \wedge (\exists\, attribut\ a : a \in droite(assoc) \\
\quad\quad\quad \wedge attribut(a,t))) \\
\quad\ \vee ((\exists\, variable\ v \in variable(droite(assoc))) \\
\quad\quad \wedge (\exists\, attribut\ a : a \in gauche(assoc) \\
\quad\quad\quad \wedge attribut(a,t))) \\
\quad \wedge (\nexists\, table\ t\ ,\ t \in TableDef \wedge \\
\quad\ ((\exists\, variable\ v \in variable(gauche(assoc)) \\
\quad\quad \wedge (\exists\, attribut\ a : a \in droite(assoc) \\
\quad\quad\quad \wedge attribut(a,t))) \\
\quad\ \vee (\exists variable\ v \in variable(droite(assoc)) \\
\quad\quad \wedge (\exists\, attribut\ a : a \in gauche(assoc) \\
\quad\quad\quad \wedge attribut(a,t)))) \\
\quad\ \wedge t \notin E) \\
\quad \Rightarrow EAssoc(assoc) = E
\end{array}
$$

Implémentation

L'outil ASF+SDF [44] permet de définir des règles d'analyse de langage sous forme de code fonctionnel, qui vont être appliquées sur un arbre syntaxique, qui contient ici les requêtes SQL et la structure des tables de la base de données. L'avantage de la programmation fonctionnelle est le mécanisme du pattern matching. L'outil ASF+SDF propose également d'autres fonctionnalités telles que le parcours descendant et ascendant des arbres syntaxiques, jusqu'à la reconnaissance d'un certain pattern. Une présentation plus complète

de cet outil est disponible dans [11] et [44]. Nous allons présenter l'implémentation de deux des règles formelles que nous avons définies précédement.

La première règle est la règle principale qui comptabilise les entrées, les sorties, etc. Le code ASF ci dessous ne présente qu'en partie cette règle. Cette différence est due à des facilités d'implémentation. La fonction ASF a, dans ce cas, déjà identifié une requête SQL de type **select-from-where**. La deuxième différence, par rapport à la règle formelle, est que la fonction ASF ne renverra pas un entier mais une liste contenant les listes des tables d'entrées, de sorties, etc.

$$
\begin{aligned}
FFP(r) \quad = \quad & (Read + Entry + Write + Exit) \\
& \mid (r \in \{query\} \\
& \wedge\, table(r) \in def\_table \\
& \wedge\, Read \;=\; \#RTable(r) \\
& \wedge\, Write \;=\; \#WTable(r) \\
& \wedge\, Entry \;=\; \#ETable(r) \\
& \wedge\, Exit \;=\; \#ExTable(r)
\end{aligned}
$$

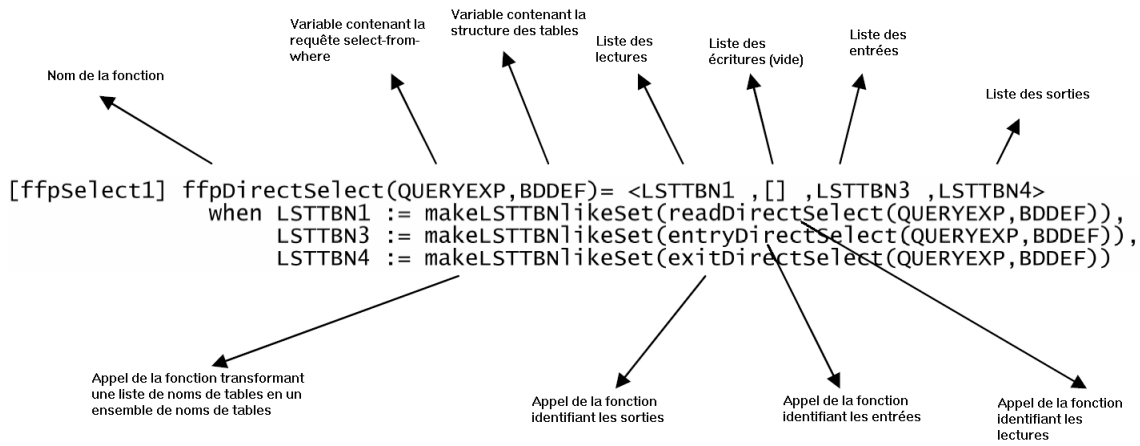

FIG. 4.5 – Définition ASF : analyse des requêtes **select-from-where**

La fonction **ffpDirectSelect** à la figure 4.5 correspond, dans les règles formelles, à la fonction **FFP**, appliquée ici à une requête de type **select-from-where**. Le mécanisme de pattern matching **ffpDirectSelect(QUERYEXP , BDDEF)**, permet de ne déclencher cette définition de la fonction **ffpDirectSelect** qu'en présence des types **QUERYEXP**, qui correspond à une requête **select-from-where** et **BDDEF**, qui contient la définition des tables. Le résultat de la fonction est <**LSTTBN1 , [] , LSTTBN3 , LSTTBN4**>, c'est-à-dire 4 listes. Ces 4 listes contiennent respectivement les noms des tables lues, des tables écrites, des tables d'entrées et des tables de sorties. Ces 4 listes sont assignées dans la clause **when**. L'assignation est représentée par le syntagme **:=**. La partie gauche de l'assignation correspond à la variable assignée. Les variables assignées sont, dans ce cas, les listes. La partie droite correspond à la valeur donnée à la variable. Dans chacune des parties droites, on retrouve la fonction technique **makeLSTTBNlikeSet** qui supprime les doublons de chaque liste, afin d'obtenir des ensembles. La fonction **readDirectSelect** correspond à la fonction **RTable** des règles formelles. De la même façon, la fonction **entryDirectSelect** correspond à la fonction **ETable** et la fonction **exitDirectSelect** représente la fonction **ExTable**. La fonction **WTable** n'a pas d'équivalent en ASF car elle renvoit toujours un ensemble vide pour une requête de type **select-from-where**. Par rapport à l'implémentation, les règles formelles restent générales et ne précisent pas le type de **r**. Par simplicité, le type de la requête

est reconnu au début d'analyse. Ces règles laissent donc des choix en ce qui concerne leur implémentation et ont plutôt pour rôle la formalisation des principes d'analyse. De même, ces règles n'entrent pas en détail dans le langage SQL par soucis de concision.

Cette deuxième règle donne les sorties d'une requête.

$$
\begin{aligned}
&ExTable(r) \equiv \\
&\quad (query(r) \Rightarrow \\
&\quad\ \forall\, attribut\ a \in clauseSelect(r) : \\
&\quad\quad \exists\, table\ t : attribut(a,t) \\
&\quad\quad \wedge\, t \in DefTable \\
&\quad\quad \Rightarrow ExTableA = \cup\{t\}) \\
&\quad \Rightarrow ExTable(r) = ExTableA
\end{aligned}
$$

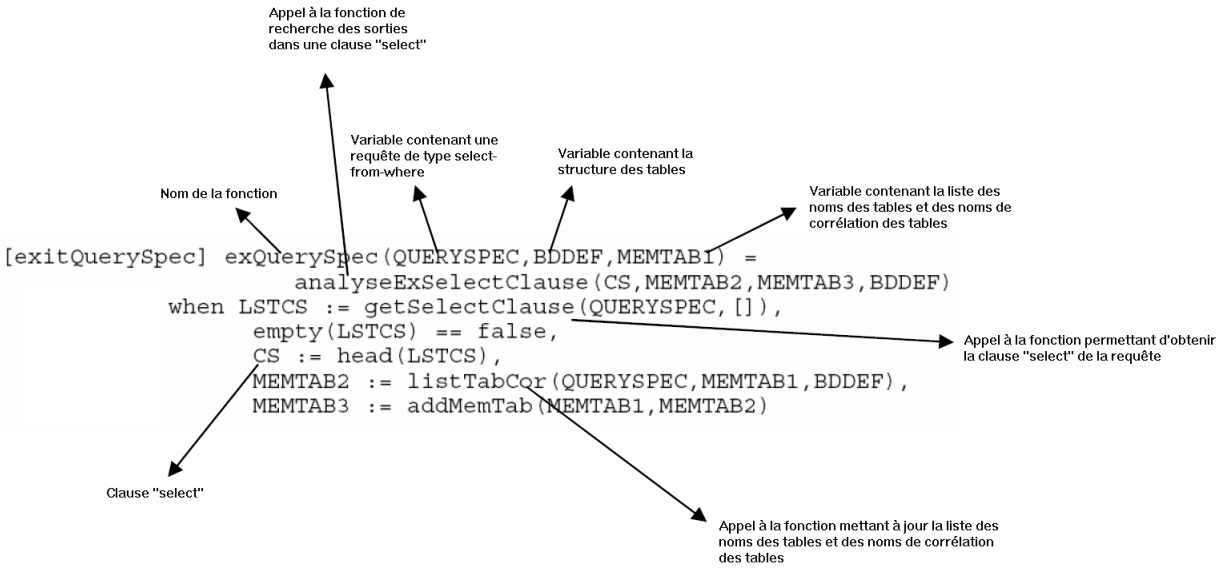

FIG. 4.6 – Définition ASF : analyse des sorties (1)

Cette fonction à la figure 4.6 lance l'analyse des sorties des requêtes. Il ne s'agit que d'une fonction de transition dans laquelle on va isoler la clause **select** et créer une liste contenant les noms des tables utilisées dans la requête, ainsi que les noms de corrélation associés aux noms de tables. Cette liste va être contenue dans les variables **MEMTAB***. Ces listes sont mises à jour par la fonction **listTabCor**. L'utilisation de plusieurs variables de ce type est dû au fait que cette fonction est également utilisée en dehors de l'analyse des sorties. Il est en effet nécessaire, dans d'autres cas, de connaître les tables de sorties d'une sous-requête. L'utilisation de deux de ces variables a permis de résoudre le problème des noms de corrélation et des noms de tables non encore déclarés. Les variables **MEMTAB*** constitue la base de connaissance utilisée par la fonction **attribut(a,t)** dans les règles formelles. L'analyse de la clause **select** va être effectuée par la fonction **analyseExSelectClause**.

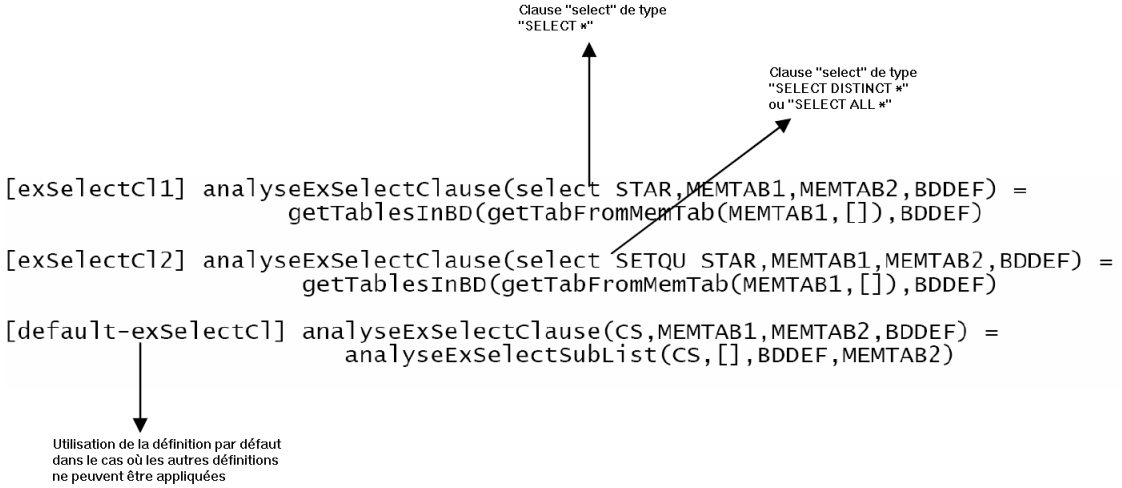

FIG. 4.7 – Définition ASF : analyse des sorties (2)

La fonction **analyseExSelectClause** à la figure 4.7 n'est ici qu'une fonction de transition qui vérifie, au moyen du pattern matching, si la clause **select** utilise le symbole *****. Ce symbole désigne toutes les tables de la clause **from** de la requête. Dans ce cas, les sorties de la requête sont les tables déclarées dans **MEMTAB1**. Dans le cas contraire, la fonction appelle **analyseExSelectSubList**.

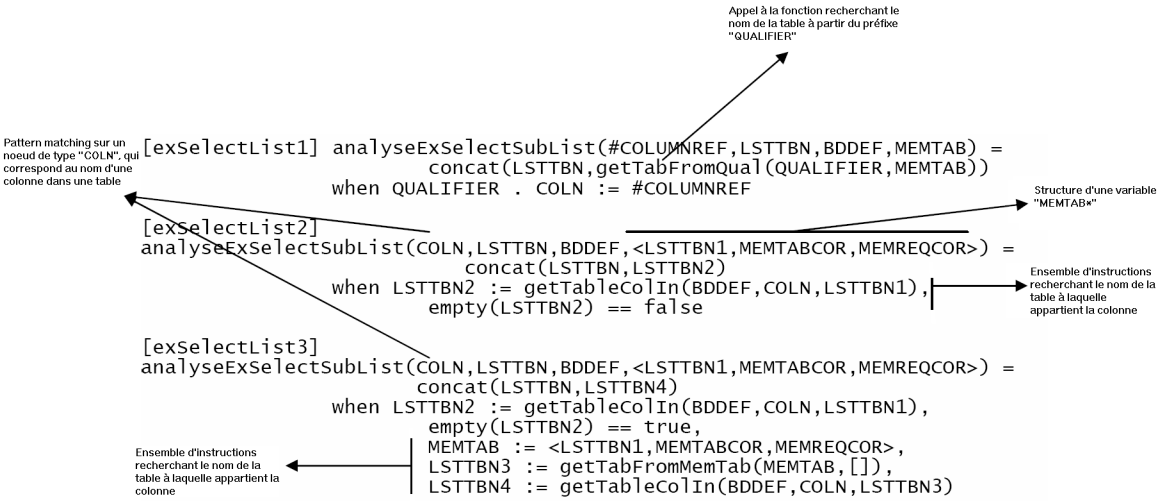

FIG. 4.8 – Définition ASF : analyse des sorties (3)

La fonction **analyseExSelectSubList** de la figure 4.8 utilise un des mécanismes proposé par l'outil ASF+SDF. Ce mécanisme permet de parcourir un arbre syntaxique automatiquement et d'identifier des noeuds d'un certain type. Par rapport à la règle formelle, le parcour descendant permet de mettre en place la partie " $\forall\, attribut\ a\ \in\ clauseSelect(r)\ :$ " de la règle. Ces noeuds sont ici représentés par le premier paramètre des 3 définitions de la fonction. Chacune des définitions représente un cas particulier à traîter en fonction du type du noeud et des conditions imposées après le **when**. La première définition cherche le nom de la table à partir du type syntaxique SQL **qualifier**. Ce type est généralement le nom d'une table ou un nom de corrélation. La deuxième définition permet de traîter directement

un nom de colonne. Ce nom doit toutefois appartenir à une table déclarée et utilisée sans un nom de corrélation.

4.1.4 Application pratique de la méthode

L'étude de cas qui va être présentée ici a été réalisée durant le stage lié à ce mémoire. Elle a été réalisée avec la collaboration du Professeur Desharnais, expert dans la mesure Cosmic-FFP. Étant donné les moyens linités dont nous disposions, cette étude reste assez restreinte et ne permet pas de valider la mesure. Le cas est le programme Open Source WebTimeEntry, disponible sur SourceForge [31]. Cette application propose une interface permettant de gérer des projets, des employés et l'attribution de ceux-ci aux projets, ainsi que les comptes et transactions des employés. Ce programme est développé en JAVA et utilise une base de données SQL. Nous avons isolé certaines fonctions de WebTimeEntry et choisi l'ensemble des processus fonctionnels touchant les employés, leurs feuilles de présence et leurs comptes. Aux trois écrans choisis, il existe dix processus fonctionnels. A ceux-ci correspondent quatorze requêtes SQL. Parmi ces requêtes, quatre n'ont pu être analysées à cause de leur construction dynamique en JAVA. Elles correspondent à deux insertions et deux mises-à-jour.

Nous avons considéré les résultats de l'expert comme référence. Ces résultats ont été obtenus à partir de la documentation du programme et des interfaces. Les résultats de l'automatisation ont été obtenus à partir des requêtes SQL utilisées par les fonctions des employés, des feuilles de présences et des comptes des employés. Les résultats se trouvent dans les tableaux aux figures 4.9, 4.10, 4.11.

Les tailles finales sont donc, pour le comptage de l'expert, écran 1 : 16 ; écran 2 : 33 ; écran 3 : 38. Les résultats de l'automatisation sont : écran 1 : 16 ; écran 2 : 29 ; écran 3 : 32. Étant donné le manque de consistance de l'étude, nous ne pouvons valider ou invalider la méthode. Nous avons pu par contre déterminer certaines des raisons pour lesquelles il existe une différence entre les deux estimations.

Nous avons observé que, lorsqu'une requête est analysable, son résultat correspond à un processus fonctionnel. Malheureusement, à un processus peuvent correspondre plusieurs requêtes redondantes dont les résultats sont équivalents en terme de mouvements de données. Là où l'expert ne verra que l'aspect graphique et donc une seule fonction, notre méthode peut utiliser des requêtes redondantes. Nous avons également constaté que si toutes les requêtes avaient été analysables, le résultat de l'automatisation aurait été bien supérieur à celui de l'expert. Afin de corriger cette différence, nous proposons de ne pas compter une requête lorsqu'elle produit les mêmes mouvements de données qu'une autre. C'est par exemple le cas avec les actions **find-It**, **Prev** et **Next** qui ne comptent que pour une seule et même vue dans l'analyse de l'expert.

4.1.5 Critiques des résultats

Les critiques ressortant de cette étude de cas sont les suivantes. La première est que les résultats de notre méthode ne sont pas trop éloignés de ceux de l'expert. De plus les variations ont pu être justifiées et peuvent être corrigées facilement. Ces variations n'ont pas été les premières observées. En effet, durant les premiers tests de la méthode, des différences sont apparues. Celles-ci étaient dues aux clés étrangères dans la base de données qui étaient considérées différemment par l'expert. Il s'est donc avéré très difficile de prendre en compte tous les paramètres d'une mesure essentiellement subjective, telle que Cosmic-FFP. Il reste sans aucun doute des erreurs non découvertes dans notre méthode.

On peut critiquer l'aspect pratique de la méthode qui demande l'identification des

Ecran :	Employee Time and Attendance
Classe :	TimeAttendance
Entité :	Employee
	employee
Nbre de requêtes pour l'écran	6 requêtes

Résultat de l'expert :

Action :	Num requête	Etat	Résultat tot.	Read	Write	Entry	Exit	Msg erreur
Find-it (lecture)	req 1	analysable	4	1	0	1	1	1
Update	req 2	non analysable	4	1	1	1	0	1
Add	req 3	non analysable	4	1	1	1	0	1
Delete	req 4	analysable	4	1	1	1	0	1
Preview (lecture)	req 6	analysable	0	-	-	-	-	-
Next (lecture)	req 5	analysable	0	-	-	-	-	-
Total FFP :	16							

Résultat de la méthode :

Action :	Num requête	Etat	Résultat tot.	Read	Write	Entry	Exit	Msg erreur
Find-it (lecture)	req 1	analysable	4	1	0	1	1	1
Update	req 2	non analysable	-	-	-	-	-	-
Add	req 3	non analysable	-	-	-	-	-	-
Delete	req 4	analysable	4	1	1	1	0	1
Preview (lecture)	req 6	analysable	4	1	0	1	1	1
Next (lecture)	req 5	analysable	4	1	0	1	1	1
Total FFP :	16							

FIG. 4.9 – Résultat de la partie ”Employee Time and Attendance”.

Ecran :	Time Card Approvals
Classe :	TimeAttendance
	TimeCard
Entité :	timeentry
Nbre de requêtes pour l'écran	2 requêtes

Résultat de l'expert :

Action :	Num requête	Etat	Résultat tot.	Read	Write	Entry	Exit	Msg erreur
Refresh (lecture)	req 7	analysable	14	6	0	1	6	1
Apply approval (update)	req 8	analysable	19	6	6	6	0	1
Total FFP :	33							

Résultat de la méthode :

Action :	Num requête	Etat	Résultat tot.	Read	Write	Entry	Exit	Msg erreur
Refresh (lecture)	req 7	analysable	14	6	0	1	6	1
Apply approval (update)	req 8	analysable	15	6	6	2	0	1
Total FFP :	29							

FIG. 4.10 – Résultat de la partie ”Time Card Approvals”.

Ecran :	Define Accruals
Classe :	TimeAttendanceAccrualRules
Entité :	accrualrules
Nbre de requêtes pour l'écran	6 requêtes

Résultat de l'expert :

Action :	Num requête	Etat	Résultat tot.	Read	Write	Entry	Exit	Msg erreur
Find-it (lecture)	req 9	analysable	8	3	0	1	3	1
Update	req 12	analysable	10	3	3	3	0	1
Add	req 11	non analysable	10	3	3	3	0	1
Delete	req 10	non analysable	10	3	3	3	0	1
Preview (lecture)	req 14	analysable	0	-	-	-	-	-
Next (lecture)	req 13	analysable	0	-	-	-	-	-
Total FFP :	38							

Résultat de la méthode :

Action :	Num requête	Etat	Résultat tot.	Read	Write	Entry	Exit	Msg erreur
Find-it (lecture)	req 1	analysable	8	3	0	1	3	1
Update	req 2	non analysable	-	-	-	-	-	-
Add	req 3	non analysable	-	-	-	-	-	-
Delete	req 4	analysable	8	3	3	1	0	1
Preview (lecture)	req 6	analysable	8	3	0	1	3	1
Next (lecture)	req 5	analysable	8	3	0	1	3	1
Total FFP :	32							

FIG. 4.11 – Résultat de la partie ”Define Accruals”.

requêtes. Cette identification peut être difficile dans le cas où elles sont construites dynamiquement. Dans notre étude, ces requête ont été signalées dans les tableaux avec le commentaire ”non analysable”. Sur 2 des 3 écrans, il existait 2 requêtes sur 6 non analysables. Cela a bien sûr influencé le résultat. De plus, nous avons supposé que le schéma conceptuel et le schéma logique étaient fort proches. Ce n'est pas toujours le cas en réalité. Il peut exister de grandes différences entre les deux suite à l'utilisation de règles de mutations vues à la section 3.1.1.

4.2 Évaluation de la cohésion

Cette section présente une application pratique de la méthode évaluant la cohésion du logiciel, présentée en 3.5.1. Comme exemple, nous avons utilisé une application boursière, qui est le résultat d'un laboratoire de méthodologie de développement de logiciel, donné en 2ème maîtrise à l'Institut d'Informatique. Pour appliquer la méthode, nous avons procédé de la façon suivante. Nous avons tout d'abord identifié les différents modules. Après identification des modules, nous avons déterminé quels étaient les accès à la base de données. Ces accès sont sous la forme de requêtes SQL. Nous avons ensuite établi différentes statistiques disponibles à l'annexe C.

La première analyse concerne la partie ”bourse”. L'application est constituée de 12 modules, correspondant à des classes JAVA et sont chargés de la partie business de l'application. Ces 12 classes se trouvent dans le package **services** de la bourse. Chacune de ces classes est chargée du traitement de certaines fonctions business, de la manière la plus indépendante possible. Parmi les 12 modules, seuls 9 avaient accès à la base de données. Les tables accédées sont au nombre de 10 et sont détaillées dans l'annexe.

Les résultats finaux sont les suivants :

Nom de la table	Nombre de modules accédant à la table	Pourcentage des modules accédant à la table
Achat	3	33,33%
Vente	3	33,33%
Broker	1	11,11%
Liquidation	1	11,11%
Historique_Jour	1	11,11%
Transaction	3	33,33%
Huissier	1	11,11%
Societe	1	11,11%
Journal_Securite	9	100,00%
Param	1	11,11%

Ces données vont nous permettre d'analyser, en partie, la cohésion de l'application. On remarque, comme énoncé dans la théorie, trois classes de tables. La première regroupe les tables accédées par un seul module. Ces tables sont : **Broker**, **Liquidation**, **Historique_Jour**, **Huissier**, **Societe** et **Param**. Ces tables représentent 60% des tables du schéma. La deuxième classe regroupe les tables techniques. La table **Journal_Securite** appartient à cette classe car elle est utilisée par chacun des 9 modules. Le dernier groupe est celui des tables indiquant un manque de cohésion. Ces tables sont : **Achat**, **Vente** et **Transaction**. Une autre observation est que les trois modules sont identiques pour les tables **Achat** et **Vente** et un seul de ces trois modules fait partie de ceux utilisant la table **Transaction**. Il s'agit du module **GestionDesTransactions**. Il y a donc 5 modules sur les 9 qui peuvent être sujets à un manque de cohésion. En pratique, le manque de cohésion nous indique un danger dans le cas où la structures des tables doit être modifiée. Il y aura, en effet, plus de modules à mettre à jour. Il nous indique également que les tâches des modules n'ont peut être pas été attribuées de façon correcte. Il pourrait donc y avoir un problème au

niveau de l'architecture du logiciel.

La deuxième analyse concerne la partie "courtier". Il y a dans cette partie, 13 modules correspondant à 13 classes JAVA situées dans le package **services** du courtier. Seuls 8 de ces modules accèdent à la base de données. Les tables utilisées sont au nombre de 10. Les résultats de l'analyse sont :

Nom de la table	Nombre de modules accédant à la table	Pourcentage des modules accédant à la table
Ordres	1	12,50%
CompteFinancier	1	12,50%
Client	1	12,50%
CompteTitre	1	12,50%
Societe_Jour	1	12,50%
Liquidation	1	12,50%
Courtier	1	12,50%
Employe	1	12,50%
Possede	1	12,50%
Journal_Securite	7	87,50%

Ces résultats nous montrent clairement que la cohésion est plus élevée dans la partie "courtier" que dans la partie "bourse". Il apparait que chaque table n'est accédée que par un seul module, à l'exception de la table **Journal_Securite**. Le rôle de cette table est de contenir une trace des événements du système. Il s'agit donc d'une table technique. Les résultats ne montrent pas de table sujette à un problème de cohérence. Cette application, contrairement à l'application boursière, possède une très bonne cohésion communicationnelle. Un processus de maintenance sur cette partie de l'application devrait donc poser moins de problèmes que sur la partie boursière.

L'analyse que nous venons de faire est simple et peu coûteuse. Toutefois, en fonction du langage de programmation utilisé, l'identification des requêtes peut s'avérer difficile, voire impossible. Dans un contexte réel, il faudra également définir la limite entre cohérent, incohérent et technique. Cette limite a été, ici, choisie arbitrairement, sur base des pourcentages obtenus. On pourrait donc définir la limite inférieure de l'incohérence à 40%. Nous pensons qu'il faudra un grand nombre d'études empiriques pour déterminer ces 2 bornes. Enfin, cette méthode ne permet pas d'évaluer la cohésion des modules n'utilisant pas la base de données. Il s'agit d'une limite liée à la définition de la méthode elle-même.

Chapitre 5

Analyse des études de cas

5.1 Observations générales faites lors des différentes mises en pratique

Les applications du chapitre 4 nous ont permis de formuler certaines critiques et observations générales.

Une première critique négative est que les outils développés et le principe même de l'étude des données et des accès aux données, tel que nous les avons abordés, limitent l'étude aux applications utilisant une base de données. Plus le nombre de modules de l'application utilisant uniquement la base de données comme espace de stockage et d'interrogation des données sera élevé, plus les mesures et méthodes développées ici permettront d'évaluer la qualité de l'application dans son ensemble. Si l'application n'a qu'un seul module utilisant la base de données, nous ne pourront évaluer la qualité que de ce module. Cela limite l'évaluation de la qualité de l'ensemble de l'application. Si par contre, tous les modules utilisent la base de données, nous pourront évaluer la qualité de ces modules et donc donner un diagnostic sur l'ensemble de l'application.

La deuxième critique négative ressortant de ces études est que les applications faites au chapitre précédent ne permettent pas de valider les mesures et méthodes utilisées. La raison est que les études présentées sont trop particulières pour permettre une validation. Des études plus complètes devront être réalisées sur des cas réels. Toutefois, les études réalisées ici nous ont permis d'identifier et de résoudre des problèmes, ainsi que d'illustrer certaines applications possibles.

Nous avons présenté, dans le chapitre 2, la méthode de Bieman et Ott concernant l'évaluation de la cohésion au moyen du slicing. Cette méthode est l'adaptation objective d'une méthode qui est à l'origine subjective. Nous pouvons tirer une conclusion similaire pour l'évaluation de la taille. La méthode développée ici permet d'approcher le résultat d'une évaluation subjective de la taille, réalisée par un expert. Plus qu'un résultat pour la maintenabilité, il s'agit d'un résultat positif sur l'ensemble du domaine des métriques. Le dialogue avec l'expert, l'étude des procédés qu'il utilise et la formalisation de ces procédés, nous ont permis de définir des règles formelles. La traduction de ces règles en un programme d'analyse nous a permis d'avoir un système automatisé, objectif et aux résultats reproductibles. Cette méthode dépend toutefois des hypothèses proposées en 3.6.2. Il s'agit là d'un point positif car l'utilisation de tels outils est moins coûteuse que l'appel à un expert. Il reste néanmoins à définir la marge d'erreur entre le jugement d'un expert et l'estimation réalisée par l'outil.

5.2 Parenthèse sur d'autres mises en pratique

Durant ce mémoire, nous avons abordé des possibilités d'utilisations que nous n'avons pu illustrer ici. Nous souhaitons donc en énoncer certaines, afin qu'elles puissent être réalisées dans d'autres recherches.

Une première application est l'étude de la complexité globale, c'est-à-dire l'étude de la différence de complexité entre du code SQL et du code procédural. Cette étude pourrait être réalisée grâce aux transformations des requêtes développées dans la partie 3.4.1. Une telle étude nous permettrait de savoir quel langage est le plus adapté dans certaines situations. De même, on pourrait y comparer les différentes implémentations des contraintes d'intégrités afin de déterminer quelles sont les plus complexes. La comparaison pourra se faire au moyen de mesures et modèles, basés sur les graphes de contrôle de flux reconnus par la communauté.

La complexité des schémas est également un point que nous aurions souhaité appliquer. Il aurait en effet été intéressant d'utiliser les mesures de taille et de complexité sur des schémas de grande taille, pouvant comporter plusieurs dizaines de types d'entités. Cette application nous aurait permis d'étudier la complexité moyenne des schémas en fonction de leur taille, et de déterminer quels sont les éléments moins maintenables dans un schéma.

Une étude de la cohésion de différents cas réels serait intéressante à réaliser. Celle-ci nous permettrait d'estimer les deux bornes définies dans la théorie. Ce type d'étude permettrait de valider ou d'invalider de façon empirique la méthode estimant la cohésion.

Une dernière étude intéressante aurait été de quantifier le lien entre le schéma des données et les mécanismes et requêtes SQL. Il est évident que certains éléments complexes du schéma sont représentés soit par du code procédural, soit par des mécanismes SQL. On peut donc établir le lien entre la complexité de ces mécanismes et celle du schéma. De même, il pourrait être intéressant d'étudier le lien entre la taille et la complexité du schéma et la complexité des requêtes. Il parait difficile de construire des requêtes complexes si le schéma lui-même n'est pas complexe. Par exemple, s'il n'existe pas de clé étrangère, il sera difficile de construire des requêtes imbriquées et des jointures. Il s'agit évidemment d'un cas extrême qui ne se voit pas en pratique. L'existence de ce lien doit donc être étudié par des études de cas.

5.3 Possibilités d'utilisation des techniques développées

Nous proposons ici des applications possibles des outils développés dans ce mémoire par rapport aux mesures et modèles existant. Nous allons donc présenter ces différentes pistes.

En ce qui concerne la complexité des requêtes, nous pouvons faire un rapprochement avec [45]. Cet article, dont nous n'avons eu connaissance que vers la fin du développement de ce mémoire, traite de la qualité des requêtes SQL. Les auteurs y définissent, entre autres, une série de mesures quantitatives sur les requêtes. Parmi ces mesures, ils proposent des mesures de la structure des requêtes, c'est-à-dire du nombre de requêtes imbriquées, de jointures et d'unions. Il nous semble intéressant de faire un rapprochement entre ces mesures et notre outil de représentation de la complexité des requêtes. Nous n'avons pas présenté, dans ce mémoire, de mesure quantitative de la complexité des requêtes. En effet, nous avons préféré nous baser sur les mesures de complexité existant, applicables au code procédural. Ces mesures ne sont utilisables que grâce à la transformation des requêtes. Les auteurs de l'article ont, par contre, développé des mesures directement applicables au code SQL. Le résultat de ces mesures est utilisé ensuite comme statistique pour l'ensemble de l'application. Dans l'article, on considère donc les éléments (sous-requêtes, jointures et

unions) séparément. Dans ce mémoire, nous regroupons chacun de ces éléments, sous une même forme, c'est-à-dire une méthode JAVA. Nous pensons que les deux méthodes sont assez différentes et qu'elles peuvent donc être utilisées conjointement et se compléter.

L'approche par rétro-ingénierie du calcul de la taille fonctionnelle a toujours été problématique lorsqu'on aborde l'implémentation d'outils [4]. Nous avons vu lors de l'application de notre méthode, qu'il ne s'agissait que d'une estimation. La raison principale est que nous n'avons pris en compte que les accès aux bases de données. Afin de compléter cette méthode, nous pouvons envisager une étude complète du logiciel, allant des interfaces jusqu'aux accès aux données. La partie manquante d'un tel outil contient donc l'analyse des interfaces jusqu'aux appels des requêtes SQL. Évidemment, cet outil est très difficilement réalisable. Il peut être même impossible à réaliser en fonction du langage du logiciel et de sa structure. Il faut en effet être capable de suivre un appel parmi l'ensemble du code. L'automatisation d'une telle méthode est donc encore un problème aujourd'hui. Nous envisageons donc plutôt un outil semi-automatisé qui demanderait le moins d'interactions possibles avec un expert pour le calcul de la taille fonctionnelle du logiciel. Nous abordons cet outil dans cette partie car il existe des outils semi-automatisés, calculant la taille fonctionnelle avant la création du logiciel. De plus, le slicing [46] pourra être utilisé afin d'analyser le code du logiciel et la valeur des variables du programme.

5.4 Développement d'un outil complet

Est-il possible de développer un outil évaluant globalement la qualité du logiciel ? Nous avons abordé un certain nombre de domaines dans ce mémoire. Ceux-ci allant du schéma des données au logiciel dans sa globalité. Nous avons laissé un certains nombres de points ouverts qui doivent être approfondis dans de futures recherches et dans des études empiriques. Malgré tout, nous pouvons dire que les outils développés dans ce mémoire ne nous permettent pas d'étudier la maintenabilité du logiciel dans son ensemble.

Nous avons présenté des utilisations d'outils de ce mémoire, qui n'appartiennent pas au domaine de la maintenabilité. Par exemple, l'évaluation de la complexité et de la taille des schémas sont plus proches de la qualité des schémas que de la maintenabilité du logiciel. L'évaluation de la taille fonctionnelle peut également n'avoir aucun rapport avec un processus de maintenance. Nous avons également présenté des pistes à développer présentant des applications qui montrent que les outils développés sont encore sous-exploités. Malgré cela, nous n'avons pu aborder des problèmes comme celui du couplage dans le logiciel ou encore la lisibilité et la concision du code dans l'ensemble du logiciel. Ces attributs du logiciel ne peuvent être étudiés au travers des données car ils sont indépendants de celles-ci.

De plus, nous n'avons pu établir l'influence des données sur l'application en général. Nous avons déterminé certains liens, comme celui entre les accès et la cohésion ou encore entre les accès et la taille du logiciel. Mais la partie pratique de ce mémoire ne nous a pas permis d'étudier plus en détail ce point. Plutôt que d'influence, nous pouvons parler plus simplement d'étude de la relation entre la base de données, les requêtes et l'application. Une étude qui reste à explorer.

Chapitre 6

Conclusions et futures orientations

6.1 Conclusions générales sur l'ensemble du mémoire

Nous avons commencé ce mémoire par une introduction aux modèles et mesures de qualité. Cette introduction nous a permis de voir, dans l'état actuel des choses, quelles étaient les théories acceptées et quels étaient en général leurs points positifs et négatifs. Nous avons également parlé des frameworks décrivant les caractéristiques d'un modèle. Par rapport à cela, nous avons pu entreprendre le développement de nouveaux modèles, mesures et outils afin de constituer un nouvel apport à la maintenabilité. Les modèles et mesures utilisant les bases de données et leurs accès étant relativement rares, nous avons dû explorer ce domaine.

Cela, nous l'avons fait dans le chapitre 3. Les résultats sont des théories ponctuelles et des outils qui peuvent avoir une application pratique. Nous avons construit ce chapitre sur l'étude de la structure des données, puis sur les accès aux données et finalement en étudiant l'application en général. Durant le développent, nous avons ciblé certains attributs particuliers du logiciel.

Les mesures de taille et de complexité des schémas peuvent être utilisées pour l'évaluation de la qualité des modèles de données. L'analyse de la complexité permet de cibler les parties trop complexes d'un schéma. Ces mesures permettent aussi de définir des statistiques quant à la complexité générale d'un schéma. Enfin, certains des éléments complexes d'un schéma demandent une implémentation particulière sous forme de mécanismes SQL ou de code procédural. Il ne s'agit plus alors uniquement de maintenabilité mais aussi de prévision des coûts d'implémentation.

L'outil de représentation de la complexité des requêtes a montré qu'il est possible d'utiliser les méthodes d'évaluation de la complexité habituelles sur une transformation des requêtes. Cette transformation est la représentation sous forme procédurale de la lecture de la requête faite par un analyste. Nous n'avons donc pas tenu compte de la représentation technique. Cette représentation nous permet, par exemple, d'appliquer la complexité cyclomatique, ou d'autres mesures et modèles de complexité du code, sur des requêtes. Nous avons d'ailleurs constaté que les requêtes SQL peuvent être aussi complexes que certaines fonctions, méthodes et procédures du reste du logiciel. L'analyse de la complexité permet donc d'obtenir, au même titre que la complexité du code procédural, des informations importantes en terme de maintenabilité.

La méthode évaluant la cohésion a donné de bons résultats dans son application pra-

tique. Sa définition est encore assez informelle et demandera un développement plus approfondi, mais comme l'a montré l'application, les résultats sont simples et facilement exploitables.

Le deuxième développement pratique important a été celui de l'estimation de la taille fonctionnelle au moyen des accès à la base de données. Cet outil est sans doute le moins proche du domaine de la maintenabilité, mais c'est également le plus abouti. Il pourra être utilisé sur des logiciels existant dont on souhaite connaître la taille fonctionnelle. Si la documentation est incomplète, il ne sera pas possible de faire appel à un expert. On pourra alors utiliser la méthode développée. De plus, celle-ci a l'avantage d'être peu coûteuse. L'estimation de la taille après création du logiciel est utile lors d'un processus de maintenance afin d'estimer les coûts.

L'application de ces outils, au chapitre 4, nous a permis de corriger certains problèmes. Nous avons pu, au chapitre 5, détailler certaines mises en pratique et utilisations des outils développés. Comme nous l'avons dit, il ne paraît pas judicieux de ne se baser que sur la partie base de données pour porter un jugement général sur la qualité de l'application. Les éléments développés sont plutôt destinés à être intégrés dans des ensembles plus vastes. Leur principal avantage est qu'ils analysent un nouvel aspect du logiciel.

En conclusion, nous pouvons avancer que la structure et les accès aux données peuvent être utilisés pour évaluer, en partie, la qualité générale des applications de base de données. Les possibilités sont nombreuses et plusieurs recherches sont encore à mener, même en dehors du domaine de la maintenabilité. Ces futures orientations vont être exposées dans la section suivante.

6.2 Futures recherches

Parmi les outils développés, nous pouvons déjà orienter la mesure de la taille et de la complexité du schéma vers l'évaluation de la qualité des schémas. En effet, ces outils permettront de dire si un schéma contient trop d'éléments complexes par rapport à un standard. Il s'agit donc plus d'une question de lisibilité et concision, donc de qualité en général, que de maintenabilité.

La mesure de la complexité des requêtes peut être utilisée en combinaison avec des méthodes telles que celles décrites dans [45]. Cet article étudie les requêtes à un niveau assez général. L'outil de transformation des requêtes développé dans ce mémoire permettrait d'étudier certaines caractéristiques des requêtes. On peut également déterminer la part d'efforts associés à la maintenance des requêtes en fonction de leur complexité. Par après, il sera possible d'estimer le coût en maintenance d'une requête par rapport au reste de l'application. Il sera également intéressant d'étudier la complexité et la lisibilité globale, c'est-à-dire de comparer les possibilités du langage SQL par rapport aux autres langages en terme d'accès aux données et d'implémentations des contraintes.

La mesure de la cohésion doit être approfondie. Elle pourrait être associée à des méthodes telles que celles développée par Yourdon et Constantine. Ces méthodes, basées sur le slicing des variables, ne tenaient pas compte des requêtes aux bases de données. On aurait alors un bon complément aux méthodes existantes évaluant la cohésion du logiciel.

La théorie sur la mesure de la taille fonctionnelle est relativement complète. Les prochaines recherches pourront porter sur l'adaptation d'autres mesures que Cosmic-FFP comme, par exemple, IFPUG. On peut également profiter des méthodes de slicing pour estimer, de façon très précise, la taille fonctionnelle d'une application en partant des accès aux données jusqu'aux interfaces utilisateurs.

A ces orientations de recherches, nous pouvons bien entendu ajouter les applications

que nous n'avons pu mettre en place dans ce mémoire, ainsi que les utilisations possibles des outils développés.

Bibliographie

[1] A. Abran, J.M. Desharnais, S. Oligny, D. St-Pierre, and C. Symons. *Manuel de mesures, Guide Cosmic d'application de la norme ISO/IEC 19761 : 2003, Version 2.2*. GÉLOG, 2003.

[2] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance : Research and Practice*, 15(2), 2003.

[3] A. J. Albrecht and J. E. Jr. Gaffney. Software function,source lines of code, and development effort prediction : A software science validation. *IEEE Trans. Software Eng.*, Nov 1983.

[4] A. April, E. Merlo, and A. Abran. A reverse ingineering approach to evaluate function point rules. In *WCRE Conference*, 1997.

[5] J.M. Bieman and L.M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, Aug. 1994.

[6] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[7] B. W. Boehm and al. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[8] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, Oct 1988.

[9] R. Cheaito, M. Frappier, S. Matwin, A. Mili, and D. Crabtree. Defining and measuring maintainability. *Technical Report, Dept. of Computer Science, University of Ottawa*, Mar 1995.

[10] Peter Pin-Shan Chen. The entity relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, pages pp. 9–36, Mar 1976.

[11] Anthony Cleve. Data-centered applications conversion using program transformations. Master's thesis, FUNDP, Institut d'informatique, Namur, Belgium, 2004.

[12] Anthony Cleve. Data reverse engineering using system dependency graphs. In *13th Working Conference on Reverse Engineering (WCRE 2006), pp. 157-166*, 2006.

[13] D. Coleman and al. Using metrics to evaluate software system maintainability. *IEEE Transactions on Software Engineering*, Aug 1994.

[14] G.E. DeYoung and G.R. Kampen. Program factors as predictors of program readability. In *COMPSAC '79, Chicago, pp. 668-673*, Nov 1979.

[15] N.E. Fenton and S.L. Pfleeger. *Software Metrics : A Rigorous and Practical Approach, Second Edition*. Paperback, 1997.

[16] W. Florac and A. William. Software quality measurement : A framework for counting problems and defects. Technical Report CMU/SEI-92-TR-022,, Carnegie Mellon, 1992.

[17] International Organization for Standardization. Iso/iec 9126-1 :2001, software engineering – product quality – part 1-4, 2001.

[18] International Organization for Standardization. Iso/iec 19761 :2003(e), software engineering - cosmic-ffp - a functional size measurement method, 2003.

[19] M. Frappier. Software metrics for predicting maintainability. Technical report, University of Ottawa, Jan 1994.

[20] M. Genero, G. Poels, E. Manso, and M. Piattini. *Defining and Validating Metrics for UML Class Diagrams*, chapter 4, pages 99–160. Metrics for Software Conceptual Models. ISBN 1-86094-497-3. IMPERIAL COLLEGE PRESS, http ://www.giro.infor.uva.es/Publications/2005/GPMP05, 2005.

[21] IFPUG Group. *Function Point Counting Practices Manual Release 4.0*. Counting Practices Committee, The international Function Point Users Group (IFPUG), 1994.

[22] Jacques Guyot. Syntaxe du langage sql92.

[23] Naji Habra, Alain Abran, Miguel Lopez, and Asma Sellami. A framework for the design and verification of software measurement methods. *Journal of Systems and Software, Special Issue on Software Process and Product Measurement*, 2007 (to appear).

[24] J-L. Hainaut. *Introduction aux bases de données relationnelles*. LIBD, Institut d'Informatique, FUNDP, Namur, Belgique, 1999.

[25] J-L. Hainaut. *Bases de données et modèles de calcul, 3ème édition*. Dunod, 2000.

[26] J-L. Hainaut. *Cours d'ingénierie des bases de données : Le Modèle Entité-Association*. LIBD, Institut d'Informatique, FUNDP, Namur, Belgique, 2002.

[27] J-L. Hainaut. *Cours d'ingénierie des bases de données : Technologie des bases de données*. LIBD, Institut d'Informatique, FUNDP, Namur, Belgique, 2002.

[28] J-L. Hainaut. *Cours d'ingénierie des bases de données : Transformation de schémas*. LIBD, Institut d'Informatique, FUNDP, Namur, Belgique, 2002.

[29] J-L. Hainaut. *ours d'ingénierie des bases de données : Les modèles logique et physique*. LIBD, Institut d'Informatique, FUNDP, Namur, Belgique, 2002.

[30] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[31] http ://sourceforge.net/. Sourceforge.net.

[32] Capers Jones. *Estimating Software Costs*. McGraw Hill, 1998.

[33] Capers Jones. Software project management practices : Failure versus success. *Software Productivity Research LLC*, Oct 2004.

[34] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman E. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, Dec 1995.

[35] Jussi Koskinen. Software maintenance costs. Technical report, Information Technology Research Institute, ELTIS-project, University of Jyväskylä, Finland, 2003.

[36] Rikard Land. Software deterioration and maintainability ? a model proposal. In *Second Conference on Software Engineering Research and Practise in Sweden (SERPS)*, Karlskrona, Sweden, Oct 2002. Blekinge Institute of Technology.

[37] REVER s.a. LIBD, Université de Namur. *Java Interface for DB-Main, Reference Manual, Version 8 Release 1, DB-Main Manual Series*. LIBD, Université de Namur, REVER s.a., 2006.

[38] M. Esperanza Manso, Marcela Genero, and Mario Piattini. *No-redundant Metrics for UML Class Diagram Structural Complexity*, volume 2681/2003. Springer Berlin / Heidelberg, 2003.

[39] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, Dec 1976.

[40] J.A. McCall, P.K. Richards, and G.F. Walters. Factors in software quality. *Rome Air Development Centre*, Dec 1977.

[41] L.M. Ott and J.J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings IEEE-CS International Metrics Symposium*, pages 71–81, Dept. of Comput. Sci., Michigan Technol. Univ., Houghton, MI, May 1993. IEEE Press.

[42] D. Stavrinoudis and al. Relation between software metrics and maintainability. In *Proceedings of the FESMA99 International Conference*, pages pp. 465–476. Federation of European Software Measurement Associations, Amsterdam The Netherlands, 1999.

[43] Hee Beng Kuan Tan, Yuan Zhao, and Hongyu Zhang. Estimating loc for information systems from their conceptual data models. In *28th International Conference on Software Engineering*, pages pp. 321–330. ACM Press, New York, NY, USA, 2006.

[44] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual, Version 1.149*. Centrum voor Wiskunde en Informatica (CWI),Amsterdam, The Netherlands, 2005.

[45] Huib van den Brink, Rob van der Leek, and Joost Visser. Quality assessment for embedded sql. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007 ¡http ://www2007.ieee-scam.org/¿)*, 2007 (to appear).

[46] M. Weiser. Program slicing. In *ICSE '81 : Proceedings of the 5th international conference on Software engineering*, pages pp. 439–449. IEEE Press, Piscataway, NJ, USA, 1981.

[47] Edward Yourdon and Larry L. Constantine. *Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Yourdon Press, 1979.

Annexe A

Règles d'analyse de la taille fonctionnelle

Cette annexe regroupe l'ensemble des règles permettant d'appliquer la méthode d'estimation de la taille fonctionnelle du logiciel décrite dans la partie 3.6 et 4.1. Ces règles sont l'application de la méthode décrite à la mesure Cosmic-FFP et couvrent une très large partie du langage repris dans les requêtes SQL. Les définitions et règles données ici contiennent également celles données en 4.1.2.

A.1 Types et constructions

A.1.1 Types COSMIC-FFP

- $Read \equiv$ *Lecture au sens COSMIC-FFP*
- $Write \equiv$ *Ecriture au sens COSMIC-FFP*
- $Entry \equiv$ *Entrée au sens COSMIC-FFP*
- $Exit \equiv$ *Sortie au sens COSMIC-FFP*

A.1.2 Types généraux SQL

- $Requete \equiv$ *Le terme requête symbolise les quatres constructions SQL générales prises en compte lors de l'analyse. Celles-ci sont* $delete$, $insert$, $query$ *et* $update$
- $delete \equiv$ *Requête SQL de type Delete-From-Where*
- $insert \equiv$ *Requête SQL de type Insert-Into*
- $query \equiv$ *Requête SQL de type Select-From-Where*
- $update \equiv$ *Requête SQL de type Update-Set-Where*

A.1.3 Représentation de la structure des données

- $DefTable \equiv$ *Définition de l'ensemble des tables devant être prises en compte lors de l'analyse, de leurs attributs et de leurs contraintes référentielles*

Constructions particulières instanciées durant l'analyse

- $association\ assoc_i \equiv$ *Couple représentant une partie ou l'ensemble d'un prédicat ou d'un case SQL et symbolisant une comparaison entre deux membres de celui-ci*

- $attribut\ att_i \equiv$ *Attribut SQL,ou encore colonne d'une table, représenté soit par le nom simple de l'attribut, soit par le nom de la table à laquelle il appartient avec le nom de l'attribut ou soit par le nom de corrélation d'une table avec le nom de l'attribut*
- $case\ ca_i \equiv$ *Construction SQL de type CaseExp*
- $modification\ modif_i \equiv$ *Couple représentant une partie ou l'ensemble d'une écriture ou d'une modification. Le membre de gauche contient le nom de la colonne et le membre de droite contient la nouvelle valeur*
- $predicat\ p_i \equiv$ *"équation" booléenne au sens SQL, utilisant les opérateurs du langage*
- $queyrSpec\ qs_i \equiv$ *Requête SQL de type* $query$ *dite simple et contenant les clauses Select, From, Where, Group by et Having*
- $resultat\ res_i \equiv$ *Construction SQL de type Result et contenant le résultat d'un CaseExp*
- $ssrequete\ ssr_i \equiv$ *Requête SQL dite complexe, comprenant les unions, jointures externes,les appels explicits de tables, etc.*
- $table\ t_i \equiv$ *Table SQL*
- $variable\ v_i \equiv$ *Paramètre d'une requête SQL, provenant du langage appelant la requête*

A.2 Description des fonctions considérées comme simples

Cette section présente les fonctions utilisées dans les règles théorique. Les fonctions ci-dessous sont considérées comme données et simples à mettre en oeuvre dans le cas d'une application pratique. Ces fonctions sont regroupées en deux classes, la première contient les fonctions permettant d'associer une table ou un attribut SQL à l'entité à laquel il correspond. La deuxième contient les fonctions permettant de parcourir et d'obtenir certaines constructions SQL comme par exemple, les $QueryExp$, $QuerySpec$, $Case$, etc.

- $attribut(a, t) \equiv$ *vrai si a est un attribut de l'entité t, faux sinon. L'évaluation se fait à partir de TableDef*
- $query(r) \equiv$ *vrai si la requête est de type Select-From-Where, faux sinon*
- $clauseFrom(r) \equiv$ *renvoit la clause From d'une requête SQL*
- $predicat(r) \equiv$ *donne l'ensemble des prédicats d'une requête*
- $association(p) \equiv$ *donne l'ensemble des associations composant le prédicat p*
- $gauche(assoc) \equiv$ *donne la partie gauche de l'association assoc*
- $droite(assoc) \equiv$ *donne la partie droite de l'association assoc*

A.3 Définitions des fonctions d'analyse

Fonction générale [1]

$$\begin{aligned} FFP(r) \quad = \quad & (Read + Entry + Write + Exit) \mid (r \in \{query, update, delete, insert\} \\ & \wedge\ table(r) \in def\_table \\ & \wedge\ Read = \#RTable(r) \\ & \wedge\ Entry = \#ETable(r) \\ & \wedge\ Write = \#WTable(r) \\ & \wedge\ Exit = \#XTable(r) + 1 \end{aligned}$$

Comptage des lectures

[1] +1 pour les sorties dans la fonction FFP car Cosmic-FFP compte un message d'erreur par défaut dans les sorties d'un processus

$$
\begin{aligned}
RTable(r) \equiv \quad & (query(r) \Rightarrow \\
& (((explicitTable(r) = \emptyset \Rightarrow E_{11} = \emptyset) \\
& \quad \vee (\exists E_1\{t_1, \ldots, t_n\} = explicitTable\_NQuerySpec\_NQueryExp(r) : \\
& \quad \exists E_{11} = \textstyle\bigcup_{i=1}^{n} tableLue(t_i))) \\
& \wedge ((querySpec(r) = \emptyset \Rightarrow E_{21} = \emptyset) \\
& \quad \vee (\exists E_2\{qs_1, \ldots, qs_m\} = querySpec\_NQueryExp(r) : \\
& \quad \exists E_{21} = \textstyle\bigcup_{i=1}^{m} RQuerySpec(qs_i))) \\
& \wedge ((ssrequete(r) = \emptyset \Rightarrow E_{31} = \emptyset) \\
& \quad \vee (\exists E_3\{ssr_1, \ldots, ssr_l\} = ssrequete\_NQuerySpec(r) : \\
& \quad \exists E_{31} = \textstyle\bigcup_{i=1}^{l} RTable(ssr_i)))) \\
& \Rightarrow RTable(r) = E_{11} \cup E_{21} \cup E_{31})) \\
& \vee (update(r) \Rightarrow RTable(r) = (RUpdate(r) \cup RSet(r) \cup RWhere(r))) \\
& \vee (delete(r) \Rightarrow RTable(r) = (RFromDel(r) \cup RWhere(r))) \\
& \vee (delete(r) \Rightarrow RTable(r) = (RInto(r) \cup RData(r)))
\end{aligned}
$$

$$
\begin{aligned}
RFom(r) \equiv \quad & \exists\, \{t_1, \ldots, t_n\}\ E\ : \\
& \quad (\forall\, table\ t_i \in E, i = 1, \ldots, n, ((t_i \in table\_NQueryExp(clauseFrom(r)) \\
& \quad \wedge\, t_i \in DefTable) \\
& \quad \vee (\exists ssrequete\ ssr\ \in\ ssrequete(clauseFrom(r))\ :\ t\ \in RTable(r)))) \\
& \wedge \nexists\, t\ :\ ((t \in table\_NQueryExp(clauseFrom(r)) \\
& \quad \wedge\, t \in DefTable \wedge t \notin E) \\
& \quad \vee (\exists ssrequete\ ssr\ \in\ ssrequete(clauseFrom(r))\ : \\
& \quad t\ \in RTable(r) \wedge\ t\ \notin E)) \\
& \Rightarrow RFrom(r)\ = \textstyle\bigcup_{i=1}^{n} tableLue(t_i)
\end{aligned}
$$

$$
\begin{aligned}
RSelect(r) \equiv \quad & (ssrequete(clauseSelect(r)) = \emptyset \\
& \quad \Rightarrow RSelect(r) = \emptyset) \\
& \vee (ssrequete(clauseSelect(r)) \neq \emptyset \\
& \Rightarrow (\exists E\{ssr_1, \ldots, ssr_n\} = ssrequete(clauseSelect(r)) \\
& \quad \wedge RSelect(r) = \textstyle\bigcup_{i=1}^{m} RTable(ssr_i)))
\end{aligned}
$$

$$
\begin{aligned}
RWhere(r) \equiv \quad & (ssrequete(clauseWhere(r)) = \emptyset \\
& \quad \Rightarrow RWhere(r) = \emptyset) \\
& \vee (ssrequete(clauseWhere(r)) \neq \emptyset \\
& \Rightarrow (\exists E\{ssr_1, \ldots, ssr_n\} = ssrequete(clauseWhere(r)) \\
& \quad \wedge RWhere(r) = \textstyle\bigcup_{i=1}^{m} RTable(ssr_i)))
\end{aligned}
$$

$$
\begin{aligned}
RSet(r) \equiv \quad & (ssrequete(clauseSet(r)) = \emptyset \\
& \quad \Rightarrow RSet(r) = \emptyset) \\
& \vee (ssrequete(clauseSet(r)) \neq \emptyset \\
& \Rightarrow (\exists E\{ssr_1, \ldots, ssr_n\} = ssrequete(clauseSet(r)) \\
& \quad \wedge RSet(r) = \textstyle\bigcup_{i=1}^{m} RTable(ssr_i)))
\end{aligned}
$$

$$
\begin{aligned}
RData(r) \equiv \quad & (ssrequete(clauseData(r)) = \emptyset \\
& \quad \Rightarrow RData(r) = \emptyset) \\
& \vee (ssrequete(clauseData(r)) \neq \emptyset \\
& \Rightarrow (\exists E\{ssr_1, \ldots, ssr_n\} = ssrequete(clauseData(r)) \\
& \quad \wedge RData(r) = \textstyle\bigcup_{i=1}^{m} RTable(ssr_i)))
\end{aligned}
$$

$$
\begin{aligned}
RHaving(r) \equiv \quad & (ssrequete(clauseHaving(r)) = \emptyset \\
& \quad \Rightarrow RHaving(r) = \emptyset) \\
& \vee (ssrequete(clauseHaving(r)) \neq \emptyset \\
& \Rightarrow (\exists E\{ssr_1, \ldots, ssr_n\} = ssrequete(clauseHaving(r)) \\
& \quad \wedge RHaving(r) = \textstyle\bigcup_{i=1}^{m} RTable(ssr_i)))
\end{aligned}
$$

$$RUpdate(r) \equiv \quad (\exists\{t_1,\ldots,t_n\}E : \exists t \in clauseUpdate(r) \\ E = tableLue(t)) \Rightarrow RUpdate(r) = E$$

$$RInto(r) \equiv \quad (\exists\{t_1,\ldots,t_n\}E : \exists t \in clauseInto(r) \\ E = tableLue(t)) \Rightarrow RInto(r) = E$$

$$RFromDel(r) \equiv \quad (\exists\{t_1,\ldots,t_n\}E : \exists t \in clauseFromDel(r) \\ E = tableLue(t)) \Rightarrow RFromDel(r) = E$$

$$RQuerySpec(r) \equiv \quad RQuerySpec(r) = \\ (RSelect(qs) \cup RFrom(qs) \cup RWhere(qs) \cup RHaving(qs))$$

Comptage des sorties

$$\begin{aligned} XTable(r) \equiv\ & (update(r) \vee delete(r) \vee insert(r) \Rightarrow XTable(r) = \emptyset) \\ & \vee(query(r) \Rightarrow \\ & ((\exists\{t_1,\ldots,t_l\}E_1 = explicitTable\_NQuerySpec\_QueryExp(r) : \\ & \quad \exists E_{11} = \textstyle\bigcup_{i=1}^{l} tableLue(t_i)) \\ & \ \vee(explicitTable\_NQuerySpec\_QueryExp(r) = \emptyset \Rightarrow E_{11} = \emptyset)) \\ & \wedge((\exists\{qs_1,\ldots,qs_m\}E_2 = querySpec\_NJoinSpec\_QueryExp(r) : \\ & \quad \exists E_{21} = \textstyle\bigcup_{i=1}^{m} XQuerySpec(qs_i)) \\ & \ \vee(querySpec\_NJoinSpec\_QueryExp(r) = \emptyset \Rightarrow E_{21} = \emptyset)) \\ & \wedge((\exists\{ssr_1,\ldots,ssr_n\}E_2 = ssrequete\_NJoinSpec\_NQuerySpec(r) : \\ & \quad \exists E_{31} = \textstyle\bigcup_{i=1}^{n} XTable(ssr_i)) \\ & \ \vee(ssrequete\_NJoinSpec(r) = \emptyset \Rightarrow E_{31} = \emptyset)) \\ & \Rightarrow XTable(r) = E_{11} \cup E_{21} \cup E_{31}) \end{aligned}$$

$$\begin{aligned} XQuerySpec(qs) \equiv\ & ((attribut\_NQueryExp\_NCaseExp(clauseSelect()) = \emptyset \Rightarrow E_{11} = \emptyset) \\ & \ \vee(\exists E_1\{att_1,\ldots,att_l\} = attribut\_NQueryExp\_NCaseExp(clauseSelect()) : \\ & \quad \forall att_i \in E_1, i = 1,\ldots,l, \exists t_i = MemTab(att_i)) \\ & \quad \wedge \exists E_{11}\{t_1,\ldots,t_m\} = \textstyle\bigcup_{k=1}^{l} TableLue(att_k,t_k)) \\ & \wedge((case\_NQueryExp(clauseSelect()) = \emptyset \Rightarrow E_{21} = \emptyset) \\ & \ \vee(\exists E_2\{ca_1,\ldots,ca_l\} = case\_NQueryExp(clauseSelect()) : \\ & \quad \wedge \exists E_{22}\{res_1,\ldots,res_m\} = \textstyle\bigcup_{k=1}^{l} resultat(ca_i)) \\ & \quad \wedge \exists E_{21}\{t_1,\ldots,t_m\} = \textstyle\bigcup_{j=1}^{m} XCaseResult(res_j)) \\ & \wedge((ssrequete\_NCaseExp(clauseSelect()) = \emptyset \Rightarrow E_{31} = \emptyset) \\ & \ \vee(\exists E_3\{ssr_1,\ldots,ssr_n\} = ssrequete\_NCaseExp(clauseSelect()) : \\ & \quad \exists E31\{t_1,\ldots,t_m\} = \textstyle\bigcup_{i=1}^{n} XTable(ssr_i))) \\ & \Rightarrow XQuerySpec(qs) = E_{11} \cup E_{21} \cup E31 \end{aligned}$$

$$\begin{aligned} XCaseResult(res) \equiv\ & ((attribut\_NQueryExp\_NCaseExp(res) = \emptyset \Rightarrow E_{11} = \emptyset) \\ & \ \vee(\exists E_1\{att_1,\ldots,att_l\} = attribut\_NQueryExp\_NCaseExp(res) : \\ & \quad \forall att_i \in E_1, i = 1,\ldots,l, \exists t_i = MemTab(att_i)) \\ & \quad \wedge \exists E_{11}\{t_1,\ldots,t_m\} = \textstyle\bigcup_{k=1}^{l} TableLue(att_k,t_k)) \\ & \wedge((case\_NQueryExp(res) = \emptyset \Rightarrow E_{21} = \emptyset) \\ & \ \vee(\exists E_2\{ca_1,\ldots,ca_l\} = case\_NQueryExp(res) : \\ & \quad \wedge \exists E_{22}\{res_1,\ldots,res_m\} = \textstyle\bigcup_{k=1}^{l} resultat(ca_i)) \\ & \quad \wedge \exists E_{21}\{t_1,\ldots,t_m\} = \textstyle\bigcup_{j=1}^{m} XCaseResult(res_j)) \\ & \wedge((ssrequete\_NCaseExp(res) = \emptyset \Rightarrow E_{31} = \emptyset) \\ & \ \vee(\exists E_3\{ssr_1,\ldots,ssr_n\} = ssrequete\_NCaseExp(res) : \\ & \quad \exists E31\{t_1,\ldots,t_m\} = \textstyle\bigcup_{i=1}^{n} XTable(ssr_i))) \\ & \Rightarrow XCaseResult(res) = E_{11} \cup E_{21} \cup E31 \end{aligned}$$

Comptage des écritures

$$
\begin{aligned}
WTable(r) \equiv \quad & (query(r) \Rightarrow WTable(r) = \emptyset) \\
& \vee (insert(r) \Rightarrow WTable(r) = RInto(r)) \\
& \vee (delete(r) \Rightarrow WTable(r) = RFromDel(r)) \\
& \vee (update(r) \Rightarrow \\
& \quad (\exists E_1 \{att_1, \ldots, att_n\} = attribut\_NUpdateSource(clauseSet(r)) \\
& \qquad \wedge \exists t \in clauseUpdate(r) : WTable(r) = \textstyle\bigcup_{i=1}^{n} tableLue(att_i, t)))
\end{aligned}
$$

Comptage des entrées

$$
\begin{aligned}
ETable(r) \equiv \quad & (query(r) \Rightarrow ETable(r) = EComparison(r) \\
& \vee (insert(r) \Rightarrow Etable(r) = EComparison(r) \cup ESet(r)) \\
& \vee (delete(r) \Rightarrow Etable(r) = EComparison(r)) \\
& \vee (update(r) \Rightarrow Etable(r) = EComparison(r) \cup EData(r))
\end{aligned}
$$

$$
\begin{aligned}
EComparison(r) \equiv \quad & ((case\_NQueryExp(r) = \emptyset \Rightarrow E_{11} = \emptyset) \\
& \vee (case\_NQueryExp(r) \neq \emptyset \Rightarrow \\
& \quad \exists E_1 \{ca_1, \ldots, ca_l\} = case\_NQueryExp(r) : \\
& \quad \exists E_{12} \{assoc_1, \ldots, assoc_m\} = \textstyle\bigcup_{i=1}^{l} association(ca_i) \\
& \quad \wedge \exists E_{11} = \textstyle\bigcup_{j=1}^{m} EAssoc(assoc_j))) \\
& \wedge ((predicat\_NQueryExp(r) = \emptyset \Rightarrow E_{21} = \emptyset) \\
& \vee (predicat\_NQueryExp(r) \neq \emptyset \Rightarrow \\
& \quad \exists E_2 \{p_1, \ldots, p_m\} = predicat\_NQueryExp(r) : \\
& \quad \exists E_{22} \{assoc_1, \ldots, assoc_n\} = \textstyle\bigcup_{i=1}^{m} association(p_i) \\
& \quad \wedge \exists E_{21} = \textstyle\bigcup_{j=1}^{n} EAssoc(assoc_j))) \\
& \wedge ((ssrequete(r) = \emptyset \Rightarrow E_{31} = \emptyset) \\
& \vee (ssrequete(r) \neq \emptyset \Rightarrow \\
& \quad \exists E_3 \{ssr_1, \ldots, ssr_n\} = ssrequete\_NCaseExp(res) : \\
& \quad \exists E_{31} = \textstyle\bigcup_{i=1}^{n} EComparison(ssr_i))) \\
& \Rightarrow EComparison(r) = E_{11} \cup E_{21} \cup E_3 1
\end{aligned}
$$

$$
\begin{aligned}
EAssoc(assoc) \equiv\ & (\exists\{t1,\ldots,t_n\}E : \\
& \forall\, table\, t\ \in\ E,\ t\ \in\ DefTable \wedge \\
& ((((\exists v \in variable\_NCaseExp\_NQueryExp(gauche(assoc))) \\
& \quad \vee (\exists ca \in case\_NQueryExp(gauche(assoc)) : \exists res \in resultat(ca) : \\
& \quad variableInResult(res) = vrai)) \\
& \wedge((\exists attributatt : att \in attribut\_NCaseExp\_NQueryExp(droite(assoc)) \\
& \quad \wedge \exists t_a \in MemTab(a) \wedge t \in tableLue(a,t_a)) \\
& \vee (\exists\, ssrequete\ ssr \in ssrequete\_NCaseExp(droite(assoc)) : t \in XTable(ssr)) \\
& \vee(\exists ca : ca \in case\_NQueryExp(droite(assoc)) \\
& \quad \wedge \exists E_2\{res_1,\ldots,res_n\} = resultat(ca) \\
& \quad \wedge t \in \textstyle\bigcup_{i=1}^{n} XCaseResult(res_i)))) \\
& \vee(\exists v\ \in\ variable\_NCaseExp\_NQueryExp(droite(assoc))) \\
& \quad \vee (\exists ca \in case\_NQueryExp(droite(assoc)) : \exists res \in resultat(ca) : \\
& \quad variableInResult(res) = vrai)) \\
& \wedge((\exists\, attribut\ att\ :\ att\ \in\ attribut\_NCaseExp\_NQueryExp(gauche(assoc)) \\
& \quad \wedge \exists t_a \in MemTab(a) \wedge t \in tableLue(a,t_a)) \\
& \vee (\exists\, ssrequete\ ssr \in ssrequete\_NCaseExp(gauche(assoc)) : t \in XTable(ssr)) \\
& \vee(\exists ca : ca \in case\_NQueryExp(gauche(assoc)) \\
& \quad \wedge \exists E_2\{res_1,\ldots,res_n\} = resultat(ca) \\
& \quad \wedge t \in \textstyle\bigcup_{i=1}^{n} XCaseResult(res_i))))) \\
& \wedge(\nexists\, table\, t\ ,\ t\ \in\ TableDef \wedge \\
& ((\exists\, variable\, v\ \in\ variable\_NCaseExp\_NQueryExp(gauche(assoc))) \\
& \quad \vee (\exists caseca \in case\_NQueryExp(gauche(assoc)) : \exists res \in resultat(ca) : \\
& \quad variableInResult(res) = vrai)) \\
& \wedge((\exists\, attribut\ att\ :\ att\ \in\ attribut\_NCaseExp\_NQueryExp(droite(assoc)) \\
& \quad \wedge \exists t_a \in MemTab(a) \wedge t \in tableLue(a,t_a)) \\
& \vee (\exists\, ssrequete\ ssr \in ssrequete\_NCaseExp(droite(assoc)) : t \in XTable(ssr)) \\
& \vee(\exists ca : ca \in case\_NQueryExp(droite(assoc)) \\
& \quad \wedge \exists E_2\{res_1,\ldots,res_n\} = resultat(ca) \\
& \quad \wedge t \in \textstyle\bigcup_{i=1}^{n} XCaseResult(res_i)))) \\
& \vee(\exists\, variable\, v\ \in\ variable\_NCaseExp\_NQueryExp(droite(assoc))) \\
& \quad \vee (\exists caseca \in case\_NQueryExp(droite(assoc)) : \exists res \in resultat(ca) : \\
& \quad variableInResult(res) = vrai)) \\
& \wedge((\exists\, attribut\ att\ :\ att\ \in\ attribut\_NCaseExp\_NQueryExp(gauche(assoc)) \\
& \quad \wedge \exists t_a \in MemTab(a) \wedge t \in tableLue(a,t_a)) \\
& \vee (\exists\, ssrequete\ ssr \in ssrequete\_NCaseExp(gauche(assoc)) : t \in XTable(ssr)) \\
& \vee(\exists ca : ca \in case\_NQueryExp(gauche(assoc)) \\
& \quad \wedge \exists E_2\{res_1,\ldots,res_n\} = resultat(ca) \\
& \quad \wedge t \in \textstyle\bigcup_{i=1}^{n} XCaseResult(res_i)))))) \\
& \Rightarrow EAssoc(assoc) = E) \\
& \vee(((variable\_NCaseExp\_NQueryExp(droite(assoc)) = \emptyset) \\
& \wedge((case\_NQueryExp(droite(assoc)) = \emptyset) \\
& \quad \vee (\forall ca \in case\_NQueryExp(droite(assoc)) : \forall res \in resultat(ca) : \\
& \quad variableInResult(res) = faux))) \\
& \wedge((variable\_NCaseExp\_NQueryExp(gauche(assoc)) = \emptyset) \\
& \wedge((case\_NQueryExp(gauche(assoc)) = \emptyset) \\
& \quad \vee (\forall ca \in case\_NQueryExp(gauche(assoc)) : \forall res \in resultat(ca) : \\
& \quad variableInResult(res) = faux))) \\
& \Rightarrow EAssoc(assoc) = \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
EModif(modif) \equiv \quad & (\exists \{t1, \ldots, t_n\}\ E\ : \\
& \forall\, table\ t\ \in\ E,\ t\ \in\ DefTable \wedge \\
& ((((\exists v \in\ variable\_NCaseExp\_NQueryExp(droite(modif))) \\
& \quad \vee (\exists ca \in case\_NQueryExp(droite(modif)) : \exists res \in resultat(ca) : \\
& \quad variableInResult(res) = vrai)) \\
& \wedge (\exists\, attribut\ att\ :\ att\ \in\ gauche(modif) \\
& \quad \wedge \exists t_a \in MemTab(a) \wedge t \in tableLue(a, t_a))) \\
& \\
& \wedge (\nexists\, table\ t\ ,\ t\ \in\ TableDef \wedge \\
& ((\exists v \in\ variable\_NCaseExp\_NQueryExp(droite(modif))) \\
& \quad \vee (\exists ca \in case\_NQueryExp(droite(modif)) : \exists res \in resultat(ca) : \\
& \quad variableInResult(res) = vrai)) \\
& \wedge (\exists\, attribut\ att\ :\ att\ \in\ gauche(modif) \\
& \quad \wedge \exists t_a \in MemTab(a) \wedge t \in tableLue(a, t_a)))) \\
& \\
& \Rightarrow EModif(modif) =\ E) \\
& \\
& \vee (((variable\_NCaseExp\_NQueryExp(droite(modif)) = \emptyset) \\
& \wedge ((case\_NQueryExp(droite(modif)) = \emptyset) \\
& \quad \vee (\forall ca \in case\_NQueryExp(droite(modif)) : \forall res \in resultat(ca) : \\
& \quad variableInResult(res) = faux))) \\
& \Rightarrow EModif(modif) = \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
ESet(r)\ \equiv \quad & \exists E\{modif_1, \ldots, modif_n\} = modification(clauseSet(r)) \\
& \Rightarrow ESet(r) = \textstyle\bigcup_{i=1}^{n} EModif(modif_i)
\end{aligned}
$$

$$
\begin{aligned}
EData(r)\ \equiv \quad & \exists E\{modif_1, \ldots, modif_n\} = modification(clauseData(r)) \\
& \Rightarrow EData(r) = \textstyle\bigcup_{i=1}^{n} EModif(modif_i)
\end{aligned}
$$

Autre fonction

$$
\begin{aligned}
variableInResult(res) \equiv \quad & ((variable\_NQueryExp\_CaseExp(res) = \emptyset \\
& \quad \wedge case\_NQueryExp(res) = \emptyset) \\
& \Rightarrow variableInResult(res) = faux) \\
& \vee ((variable\_NQueryExp\_CaseExp(res) = \emptyset \\
& \quad \wedge \exists E\{ca_1, \ldots, ca_n\} = case\_NQueryExp(res) : \\
& \quad \exists E_2\{res_1, \ldots, res_m\} = \textstyle\bigcup_{i=1}^{n} resultat(ca_i)) \\
& \Rightarrow variableInResult(res) = \textstyle\bigcup_{j=1}^{m} variableInResult(res_j)) \\
& (variable\_NQueryExp\_CaseExp(res) \neq \emptyset \\
& \Rightarrow variableInResult(res) = vrai)
\end{aligned}
$$

Annexe B

Taille et complexité du schéma

B.1 Schéma d'exemple

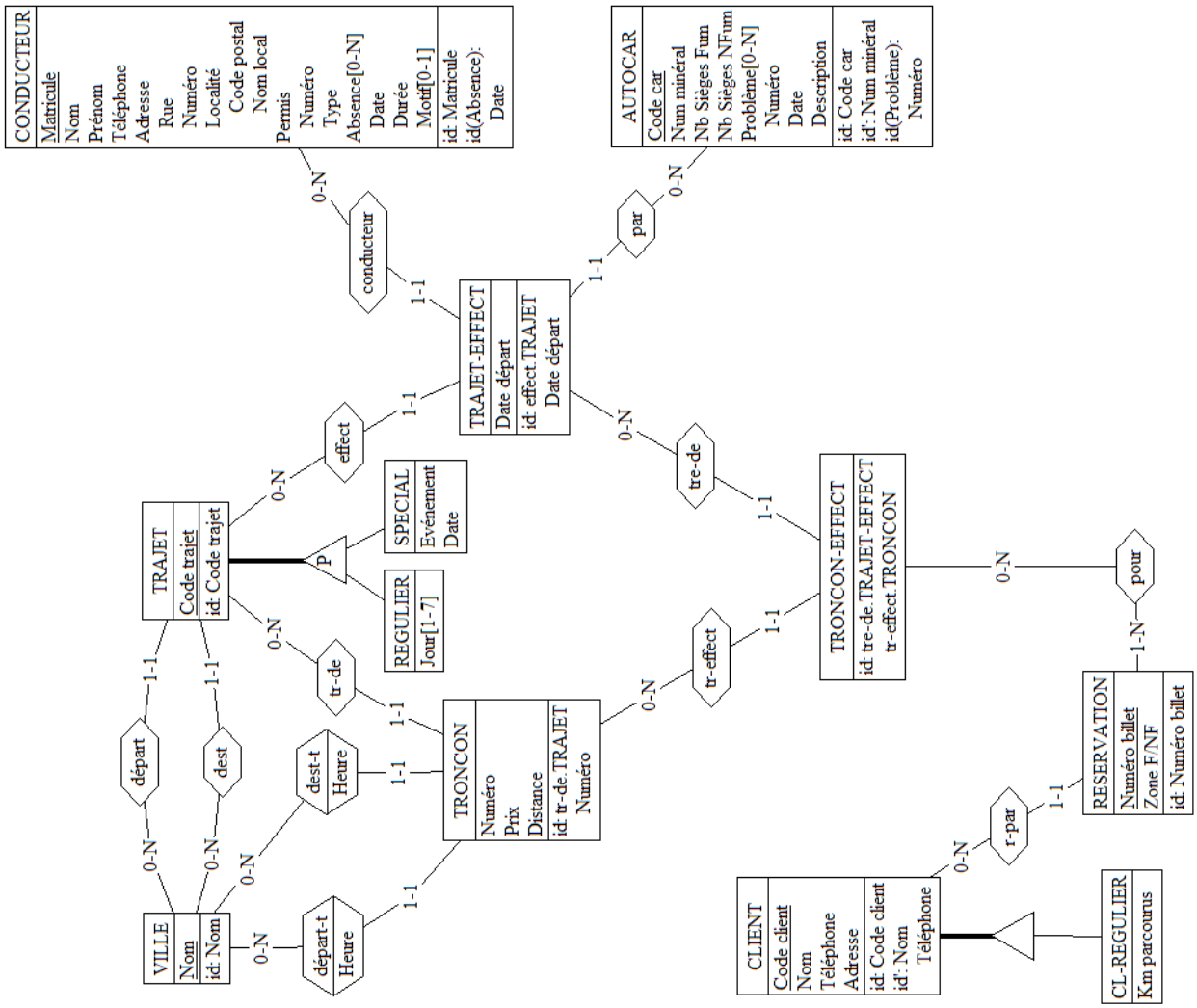

FIG. B.1 – Exemple de schéma tiré de DB-MAIN

Annexe C

Étude de la cohésion : Documents et statistiques

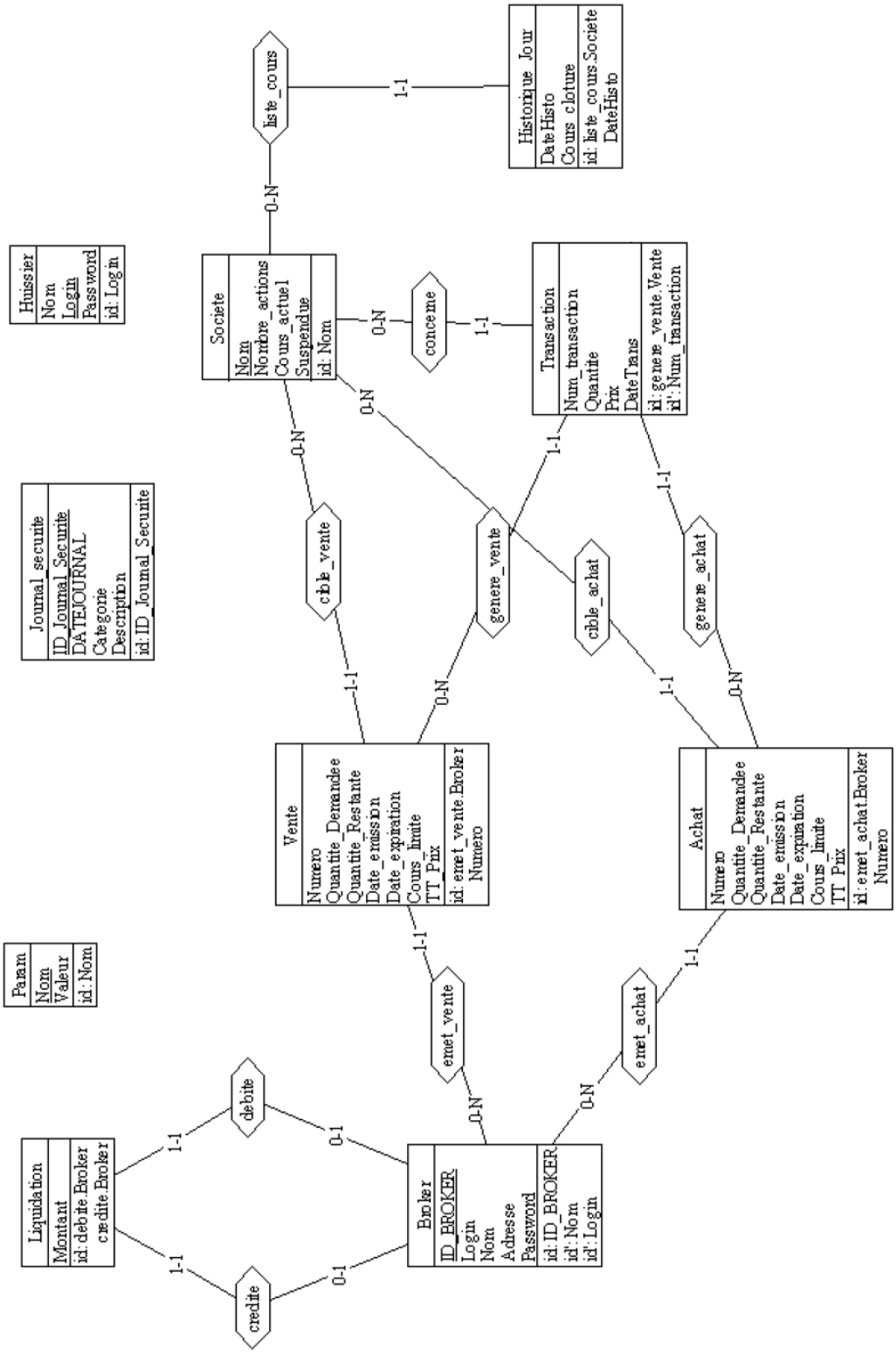

FIG. C.1 – Schéma conceptuel de la partie ”bourse”

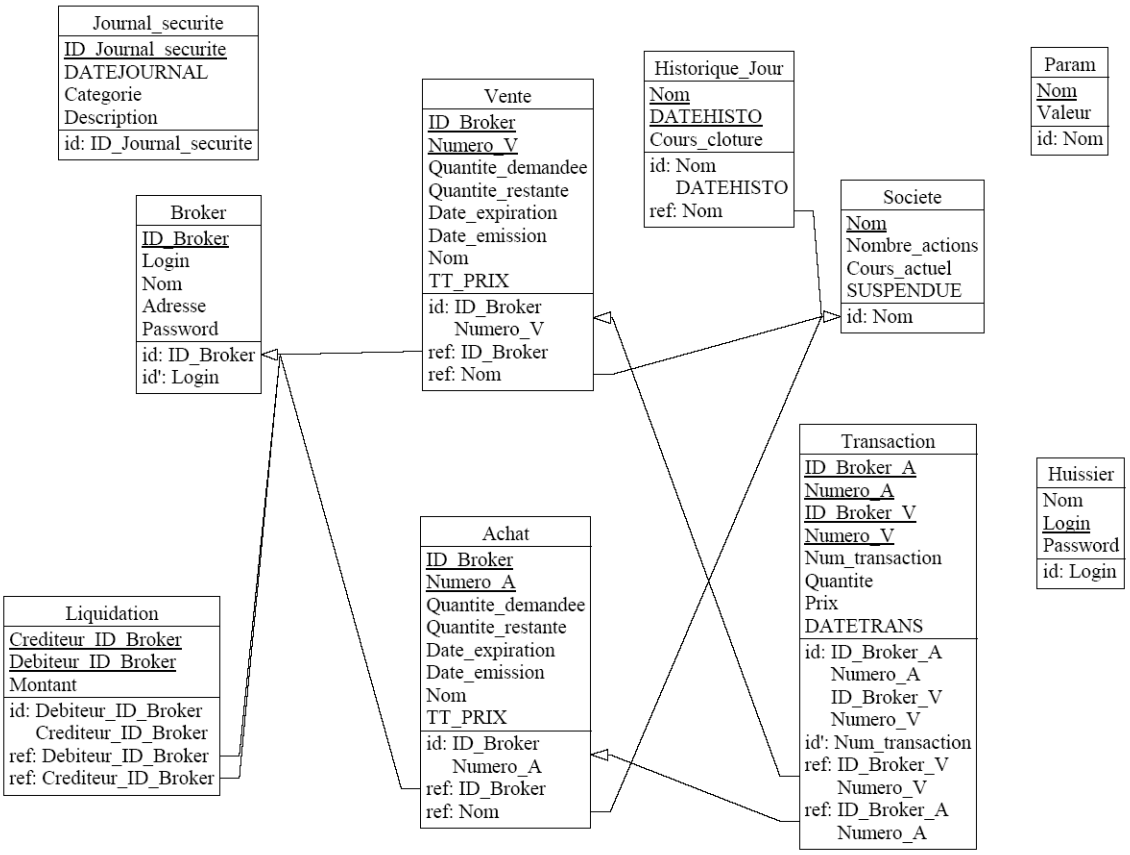

FIG. C.2 – Schéma relationnel de la partie ”bourse”

Type d'accès	SQLSelect		SQLQuery		Log	
Nom du module	Nombre d'accès	Tables concernées	Nombre d'accès	Tables concernées	Nombre d'accès	Tables concernées
ConsultationPolitique	0		0		0	Journal_Securite
ControleurTache	0		0		0	Journal_Securite
ExecuterTaches	4	Vente,Vente,Achat,Achat	4	Vente,Vente,Achat,Achat	2	Journal_Securite
GestionDesCourtiers	4	Broker, Liquidation,broker, liquidation	2	Broker,Liquidation	5	Journal_Securite
GestionDesHistoriques	2	Historique_Jour, Transaction	2	Historique_Jour,Historique_Jour	2	Journal_Securite
GestionDesHuissiers	1	Huissier	1	Huissier	2	Journal_Securite
GestionDesOrdres	11	Achat,Vente,Achat,Vente,Achat,Vente	4	Achat,Vente,Achat,Vente	3	Journal_Securite
GestionDesSocietes	1	Societe	2	Societe,Societe	6	Journal_Securite
GestionDesTransactions	6	Achat,Vente,Achat,Vente,Achat,Vente	4	transaction,transaction,transaction,transaction	3	Journal_Securite
GestionEvenements	0		0		0	Journal_Securite
GestionIdentites	0		0		2	Journal_Securite
PolitiqueBourse	2	Journal_Securite,param	1	Journal_Securite	0	Journal_Securite

Tableau des nombres d'accès effectués au moyen de SQLSelect et SQLQuery par module

Nom de la table	Achat	Vente	Broker	Liquidation	Historique_Jour	Transaction	Huissier	Societe	Journal_Securite	param
Nom du module										
ConsultationPolitique	0	0	0	0	0	0	0	0	0	0
ControleurTache	0	0	0	0	0	0	0	0	0	0
ExecuterTaches	4	4	0	0	0	0	0	0	0	0
GestionDesCourtiers	0	0	3	3	0	0	0	0	0	0
GestionDesHistoriques	0	0	0	0	3	1	0	0	0	0
GestionDesHuissiers	0	0	0	0	0	0	2	0	0	0
GestionDesOrdres	7	7	0	0	0	1	0	0	0	0
GestionDesSocietes	0	0	0	0	0	0	0	3	0	0
GestionDesTransactions	3	3	0	0	0	4	0	0	0	0
GestionEvenements	0	0	0	0	0	0	0	0	0	0
GestionIdentites	0	0	0	0	0	0	0	0	0	0
PolitiqueBourse	0	0	0	0	0	0	0	0	2	1

FIG. C.3 – Statistiques obtenues par l'analyse des modules (1)

Tableau des nombres d'accès effectués au moyen de Log par module et table

Nom du module \ Nom de la table	Achat	Vente	Broker	Liquidation	Historique_Jour	Transaction	Huissier	Societe	Journal_Securite	param	
ConsultationPolitique		0	0	0	0	0	0	0	0	0	0
ControleurTache		0	0	0	0	0	0	0	0	0	0
ExecuterTaches		0	0	0	0	0	0	0	0	2	0
GestionDesCourtiers		0	0	0	0	0	0	0	0	5	0
GestionDesHistoriques		0	0	0	0	0	0	0	0	2	0
GestionDesHuissiers		0	0	0	0	0	0	0	0	2	0
GestionDesOrdres		0	0	0	0	0	0	0	0	3	0
GestionDesSocietes		0	0	0	0	0	0	0	0	6	0
GestionDesTransactions		0	0	0	0	0	0	0	0	3	0
GestionEvenements		0	0	0	0	0	0	0	0	0	0
GestionIdentites		0	0	0	0	0	0	0	0	2	0
PolitiqueBourse		0	0	0	0	0	0	0	0	0	0

Tableau du total des nombres d'accès par module et table

Nom du module \ Nom de la table	Achat	Vente	Broker	Liquidation	Historique_Jour	Transaction	Huissier	Societe	Journal_Securite	param	
ConsultationPolitique		0	0	0	0	0	0	0	0	0	0
ControleurTache		0	0	0	0	0	0	0	0	0	0
ExecuterTaches		4	4	0	0	0	0	0	0	2	0
GestionDesCourtiers		0	0	3	3	0	0	0	0	5	0
GestionDesHistoriques		0	0	0	0	3	1	0	0	2	0
GestionDesHuissiers		0	0	0	0	0	0	2	0	2	0
GestionDesOrdres		7	7	0	0	0	1	0	0	3	0
GestionDesSocietes		0	0	0	0	0	0	0	3	6	0
GestionDesTransactions		3	3	0	0	0	4	0	0	3	0
GestionEvenements		0	0	0	0	0	0	0	0	0	0
GestionIdentites		0	0	0	0	0	0	0	0	2	0
PolitiqueBourse		0	0	0	0	0	0	0	0	2	1

FIG. C.4 – Statistiques obtenues par l'analyse des modules (2)

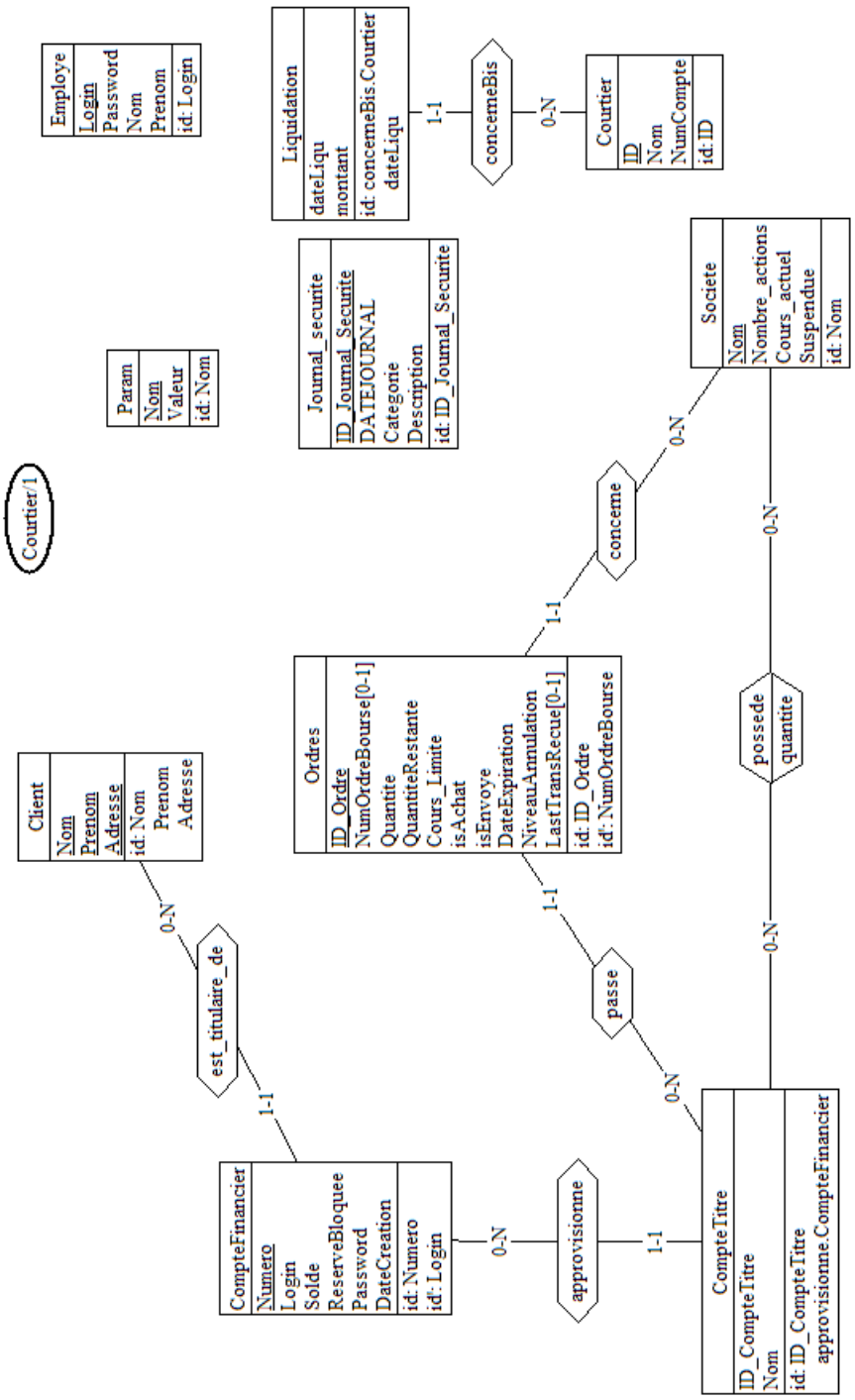

FIG. C.5 – Schéma conceptuel de la partie ”courtier”

1

Type d'accès	SQLSelect		SQLQuery		Log	
Nom du module	Nombre d'accès	Tables concernées	Nombre d'accès	Tables concernées	Nombre d'accès	Tables concernées
GestionActionsSurOrdres	4	Ordres,Ordres,Ordres,Ordres	4	Ordres,Ordres,Ordres,Ordres	9	Journal_Securite
GestionCompteClient	4	CompteFinancier,Client,CompteFinancier,CompteTitre	7	Ancier,CompteTitre,Client,CompteFinancier,CompteTitre,CompteFinancier,Compte	9	Journal_Securite
GestionConnexion	0		0		2	Journal_Securite
GestionConsultationSocietes	1	Societe	2	Societe,Societe	2	Journal_Securite
GestionDesLiquidations	1	Liquidation	2	Courtier,Liquidation	0	
GestionEmployes	1	Employe	2	Employe,Employe	5	Journal_Securite
GestionPolitiqueCourtier	1	Journal_Securite	1	Journal_Securite	4	Journal_Securite
GestionPorteFeuille	1	Possede	2	Possede,Possede	1	Journal_Securite
GestionRelationBourse	0		0		0	
LogiqueServiceClient	0		0		0	
ReceptionInfoBourse	0		0		0	
SurveillanceConnexions	0		0		0	
ThreadEvent	0		0		0	

Tableau des nombres d'accès effectués au moyen de SQLSelect et SQLQuery par module et table

Nom de la table	Ordres	CompteFinancier	Client	CompteTitre	Societe	Liquidation	Courtier	Employe	Possede	Journal_Securite
Nom du module										
GestionActionsSurOrdres	8	0	0	0	0	0	0	0	0	0
GestionCompteClient	0	5	2	3	0	0	0	0	0	0
GestionConnexion	0	0	0	0	0	0	0	0	0	0
GestionConsultationSocietes	0	0	0	0	3	0	0	0	0	0
GestionDesLiquidations	0	0	0	0	0	2	0	1	0	0
GestionEmployes	0	0	0	0	0	0	0	0	3	0
GestionPolitiqueCourtier	0	0	0	0	0	0	0	0	0	2
GestionPorteFeuille	0	0	0	0	0	0	0	0	3	0
GestionRelationBourse	0	0	0	0	0	0	0	0	0	0
LogiqueServiceClient	0	0	0	0	0	0	0	0	0	0
ReceptionInfoBourse	0	0	0	0	0	0	0	0	0	0
SurveillanceConnexions	0	0	0	0	0	0	0	0	0	0
ThreadEvent	0	0	0	0	0	0	0	0	0	0

FIG. C.6 – Statistiques obtenues par l'analyse des modules (1)

Tableau des nombres d'accès effectués au moyen de Log par module et table

Nom de la table / Nom du module	Ordres	CompteFinancier	Client	CompteTitre	Societe	Liquidation	Courtier	Employe	Possede	Journal_Securite
GestionActionsSurOrdres	0	0	0	0	0	0	0	0	0	9
GestionCompteClient	0	0	0	0	0	0	0	0	0	9
GestionConnexion	0	0	0	0	0	0	0	0	0	2
GestionConsultationSocietes	0	0	0	0	0	0	0	0	0	2
GestionDesLiquidations	0	0	0	0	0	0	0	0	0	0
GestionEmployes	0	0	0	0	0	0	0	0	0	5
GestionPolitiqueCourtier	0	0	0	0	0	0	0	0	0	4
GestionPorteFeuille	0	0	0	0	0	0	0	0	0	1
GestionRelationBourse	0	0	0	0	0	0	0	0	0	0
LogiqueServiceClient	0	0	0	0	0	0	0	0	0	0
ReceptionInfoBourse	0	0	0	0	0	0	0	0	0	0
SurveillanceConnexions	0	0	0	0	0	0	0	0	0	0
ThreadEvent	0	0	0	0	0	0	0	0	0	0

Tableau du total des nombres d'accès par module et table

Nom de la table / Nom du module	Ordres	CompteFinancier	Client	CompteTitre	Societe	Liquidation	Courtier	Employe	Possede	Journal_Securite
GestionActionsSurOrdres	8	0	0	0	0	0	0	0	0	9
GestionCompteClient	0	5	2	3	0	0	0	0	0	9
GestionConnexion	0	0	0	0	0	0	0	0	0	2
GestionConsultationSocietes	0	0	0	0	3	0	0	0	0	2
GestionDesLiquidations	0	0	0	0	0	2	1	0	0	0
GestionEmployes	0	0	0	0	0	0	0	3	0	5
GestionPolitiqueCourtier	0	0	0	0	0	0	0	0	0	6
GestionPorteFeuille	0	0	0	0	0	0	0	0	3	1
GestionRelationBourse	0	0	0	0	0	0	0	0	0	0
LogiqueServiceClient	0	0	0	0	0	0	0	0	0	0
ReceptionInfoBourse	0	0	0	0	0	0	0	0	0	0
SurveillanceConnexions	0	0	0	0	0	0	0	0	0	0
ThreadEvent	0	0	0	0	0	0	0	0	0	0

FIG. C.7 – Statistiques obtenues par l'analyse des modules (2)